NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SURVIVABILITY AS A TOOL FOR EVALUATING OPEN SOURCE SOFTWARE

by

David J. Cummings

June 2015

Thesis Advisor: Timothy H. Chung
Second Reader: Rama D. Gehris

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	06-19-2015	Master's Thesis 09-01-2014 to 06-19-2015

4. TITLE AND SUBTITLE

SURVIVABILITY AS A TOOL FOR EVALUATING OPEN SOURCE SOFTWARE

5. FUNDING NUMBERS

6. AUTHOR(S)

David J. Cummings

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis explores the application of traditional survivability analysis to the open source software (OSS) development process. It postulates that combat systems face potential threats from cyber warfare professionals aiming to manipulate software embedded in the systems. The research highlights current Department of Defense (DOD) interest in OSS, and explains a method for evaluating the capability of OSS programs to withstand cyber warfare attacks. Survivability concepts are demonstrated in a scenario involving an adversary inserting malicious code into the source repository of FlightGear, an open source flight simulator. Analysis is conducted on five open source programs to illustrate commonality in the evaluation process. It is determined that survivability analysis is a feasible method for OSS software evaluation, and could be used as a tool to compare OSS alternatives.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software, survivability, open source software, unmanned aerial vehicle	109
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SURVIVABILITY AS A TOOL FOR EVALUATING OPEN SOURCE SOFTWARE

David J. Cummings
Lieutenant, United States Navy
B.S., California Polytechnic University, San Luis Obispo, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2015

Author: David J. Cummings

Approved by: Timothy H. Chung
Thesis Advisor

Rama D. Gehris
Second Reader

Clifford Whitcomb
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the application of traditional survivability analysis to the open source software (OSS) development process. It postulates that combat systems face potential threats from cyber warfare professionals aiming to manipulate software embedded in the systems. The research highlights current Department of Defense (DOD) interest in OSS, and explains a method for evaluating the capability of OSS programs to withstand cyber warfare attacks. Survivability concepts are demonstrated in a scenario involving an adversary inserting malicious code into the source repository of FlightGear, an open source flight simulator. Analysis is conducted on five open source programs to illustrate commonality in the evaluation process. It is determined that survivability analysis is a feasible method for OSS software evaluation, and could be used as a tool to compare OSS alternatives.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ARSENL Advanced Robotic Systems Engineering Laboratory

ACS Aerial Combat Swarms

AAA anti-aircraft artillery

API application program interface

CIO Chief Information Officer

CSV comma-separated values

CAC Common Access Card

CVE Computer Vulnerabilities and Exposures

CRUSER Consortium for Robotics and Unmanned Systems Education and Research

C2 Command and Control

COTS commercial-off-the-shelf

DARPA Defense Advanced Projects Research Agency

DOD Department of Defense

E3A Essential Events and Elements Analysis

GUI graphical user interface

GCS ground control station

HTTP Hyper Text Transfer Protocol

IT information technology

IEEE Institute of Electrical and Electronics Engineers

ISR intelligence surveillance and reconnaissance

IOC initial operating capability

IADS integrated air defense system

JSF Joint Strike Fighter

LFTE Live-Fire Test and Evaluation

MCWL Marine Corps Warfighting Laboratory

MIT Massachusetts Institute of Technology

MASINT measurement and signature intelligence

NPS Naval Postgraduate School

ONR Office of Naval Research

OPNAV Office of the Chief of Naval Operations

OpenBSD Open Berkeley Software Distribution

OSI Open Source Initiative

OSS open source software

PACFLT Pacific Fleet

SLOC source lines of code

SEA South East Asia

SRWBR short range wide band radio

SAM surface-to-air missile

TCP transmission control protocol

URL uniform resource locator

U.S. United States

USCG United States Coast Guard

USG United States government

UAS unmanned aerial system

UAV unmanned aerial vehicles

UDP user datagram protocol

WWII World War II

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

The Department of Defense (DOD) relies on software to execute important strategic mission requirements. Combat systems employed by the DOD are becoming progressively more dependent on software to be effective, as evidenced by some aircraft whose engines cannot be started without a software card relaying mission information to onboard computers. The DOD has many uses for software, so it is important for the organization to make informed decisions when choosing between software alternatives. A poor choice of software could have significant implications on the budget, the readiness, and the capability of the armed services.

One category of software receiving special attention is open source software. In 2009, the DOD Chief Information Officer (CIO) reiterated that OSS should be evaluated along with proprietary software when choosing alternatives for DOD applications. Open source software is highly suitable for rapid prototyping and experimentation, and low upfront cost and availability of source code make OSS ideal for DOD research and development. The idea of OSS being embedded in operational combat systems is troubling to some because public access of open source code allows anyone, including enemies, to gain insight to system capabilities, but proprietary software may not be as secure as one might think, and OSS not as vulnerable. This thesis aims to evaluate the ability of the OSS development process to guard against attackers by determining the suitability of individual OSS programs for DOD missions.

The scope of the thesis is narrowed by studying DOD OSS programs embedded in hardware systems, specifically software utilized by small UAVs. At the Naval Postgraduate School (NPS), Advanced Robotic Systems Engineering Laboratory (ARSENL) supports ongoing research involving UAVs and UAV swarm operations, and OSS is an essential part of their operating scheme. Several OSS programs used by the lab are taken as a sample and analyzed to evaluate their survivability. Initial data gathering from one program, FlightGear, provides information to demonstrate how survivability applies to the OSS development process. Data from several other programs is retrieved using a separate software program developed for the research.

Survivability analysis is demonstrated by designing a scenario in which an adversary inserts malicious software to an OSS program by gaining access to the community of software developers building the code. Nine steps outline events allowing the adversary to successfully compromise UAV autopilot software, represented by FlightGear, and kill an aircraft. The goal of the survivability analysis is to identify the likelihood of events in the scenario occurring, and subsequently, the combined probability that aircraft is killed. In the case of the FlightGear scenario, event probabilities are determined by analyzing characteristics of FlightGear's open source project, including activity level of the development community as judged by issue reports and commit logs. This data represents one indicator of the general health of FlightGear and its resistance to attack.

One underlying theory of the thesis is that data associated with open source projects reflects their resilience against cyberattacks, and by this premise, the first step in evaluating an OSS program is to gather information about it. Many open source software programs are hosted by websites such as GitHub, GitLab, BitBucket, and SourceForge, which act as cloud-based repositories, storing the project's source code online and allowing simultaneous collaboration by developers in separate geographic locations. The websites track and archive activities related to development of programs and report information such as the number of people working on the code, the number of inputs each programmer provides, the unresolved bugs in the program, and the programming languages used for development. Obtaining this information manually by scrolling through websites and clicking on links is not efficient, so a software program is developed to automatically access project data and store it for analysis. The program is written in the Python programming language, and uses several tools to gather data from GitHub, where most of the applications used by ARSENL are kept. Time constraints for the thesis limited the program development, so it is only able to process project issues (bugs or feature requests), which is an important metric for OSS programs, but should not be the only data included in OSS analysis. Plans are laid out for development of a more thorough program in Section 5.2.

Collected data is graphically analyzed using the Microsoft Excel spreadsheet program. Specific indications of project health come from trends in data, such as the number of issues opened over the time period analyzed. An increasing trend in opened issues means there is high project activity and many developers working on the project. This is a healthy

sign, showing interest in the project and support from the community. A malicious code insertion will have a difficult time surviving inspection by many programmers working on the project, therefore, an increasing trend in issues being opened suggests better resilience to attack. This logic is applied to the survivability analysis by reducing event probabilities, which increases the likelihood that such a program would survive a malicious insertion attempt. In this way, survivability analysis successfully demonstrates its usefulness for evaluating OSS programs.

Upon completing the thesis, it was apparent there are many avenues for further research. While survivability analysis is demonstrated to work for a small sample of OSS programs, it needs to be validated by expanding its scope through application to other program types. The outcomes of the survivability analysis also need to be verified by cross-checking results with those produced by known OSS validation techniques. In addition, the program developed for the thesis needs to be expanded to capture more data. It could also be improved to perform data analysis instead of just data collection. It would be helpful for people choosing between OSS alternatives to see immediate results about an OSS program's suitability for a specific application by inputting a project name and receiving immediate results from the data analysis program. In its current state, collected data requires manual analysis using a different spreadsheet program. These topics could all supply opportunities for further research to provide better understanding of the potential role of survivability analysis in OSS software development and selection.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I thank my thesis advisor, Tim Chung, for providing major support throughout my research. He was consistently available to answer questions and determined to provide the tools necessary for me to succeed. Professor Chung was especially helpful in the development of a software program that was crucial to the completion of my thesis. Thanks to my second reader, Rama Gehris, who provided insight about software topics and offered excellent advice for the flow and structure of the document. Thanks to Naval Postgraduate School (NPS) staff working in the Advanced Robotic Systems Engineering Laboratory (ARSENL) lab, including Michael Day and Michael Clement, who provided information about open source software (OSS), and to students in the thesis working group, who made the daily grind of thesis work—especially toward the end—enjoyable and entertaining.

Thanks to the professors and faculty of the Systems Engineering Department and Mechanical Engineering Department at NPS, especially Chris Adams, Matt Boensel, Ron Giachetti, and Mark Stevens, who provided the academic background and analytic tools necessary to progress through my research. Special thanks to Barbara Berlitz for her instruction about the thesis process at NPS and her detailed review of my finished product.

HUGE thanks to my beautiful wife, Chelsea, and our two daughters, Lyla and Emmy, for empathizing with me, for remaining supportive when I could not be home, and for providing constant encouragement as I proceeded through the research process. Thank you!

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Background

Software had its origin in the middle of the 20th century, and since then software development has expanded rapidly. The Department of Defense (DOD) uses software extensively and relies on software-based systems for most of its critical capabilities. In order for United States (U.S.) military services to maintain a tactical advantage, they must stay at the leading edge of essential technology areas, including software development. One form of software that has received increasing attention in the last decade is open source software (OSS). As an example, about three-quarters of the world's large companies planned to add Linux-based servers to their information technology systems over the past 10 years [1]. The OSS software community has the support of commercial companies interested in utilizing open source approaches to develop new technology. For example, one illustrative open source project whose organizational structure is governed by the Linux Foundation, "Dronecode," aims to standardize unmanned aerial vehicles (UAV) software and promote cooperation among developers for consumer and commercial UAV applications [2].

Due to advances in OSS development and backing from major commercial companies, the government should expand its interest in OSS programs and OSS methodology. Although open source software is currently used in the DOD, apprehension sometimes blocks its implementation in sensitive areas such as combat system development. OSS programs need to be analyzed to support DOD understanding and awareness of their potential to improve technology used by U.S. military forces.

Like consumer software, military software technology has expanded rapidly in recent years, largely due to the advancing complexity of military systems. The F-22 Raptor, which demonstrated initial operating capability (IOC) in 2005, employs around two million lines of code to operate its avionics system [3]. In contrast, the F-35 Joint Strike Fighter (JSF), with an anticipated IOC date before December 2016, already has 24 million lines of code in its software system [4]. Software required to support the advanced capability of the newer

fighter is constantly forced to keep pace as hardware technology improves. Over 10 years' time, the demand for software grew by 10 times the original amount, and this trend will likely continue for future aircraft designs. Such developments highlight the DOD need for acquiring new software and understanding how it will continue to impact combat systems in the future.

While OSS may seem unconventional or even out of place in the traditional military acquisition landscape, it may be more suitable than proprietary software for certain DOD applications. In fact, in a memorandum published by the chief information officer of the DOD, OSS is designated as "commercial computer software that shall be given appropriate statutory preference" when considering software for any DOD function [5]. As such, this research aims to highlight characteristics of the OSS development process and evaluate OSS projects based on data gathered from their open source repositories.

1.2 Scope

The scope of this thesis involves military hardware systems that incorporate software in their designs. Generally, the systems in question are heavily dependent on software and would not be useful without their software-based elements. Examples are modern military aircraft, air defense systems, ships, radar systems, unmanned underwater systems and unmanned aerial vehicles. The advent of OSS in these classes of assets or systems is rare, which motivates this thesis to explore its potential capability and viability within this scope.

The primary system of focus for this thesis is the unmanned aerial vehicles. In addition to the surge in commercial development and military use in the past decade, significant work has been conducted by software developers and hobbyists alike who generate, implement, and deploy code used by small civilian UAVs and drones. This technology could potentially be very useful for military initiatives working on developing aligned systems for the military. One goal of the thesis is to assess whether OSS is feasible for use in military systems such as UAVs. The same methods used here are anticipated to be applicable to other OSS software under consideration for use by the military. Ideally, these insights may provide an analytic framework to generate guidance for decision makers that may support the inclusion of OSS to more programs of interest within the DOD.

This research does not, however, seek to directly compare proprietary software and open source software in depth. While general aspects are discussed, no formal evaluation is performed to promote or devalue one development method over another. There is potential for a comparative study; however, it is not the focus of this report, as such assessments are largely dictated by the specific requirements on the software system and/or operational context. Chapter 5 discusses options for further exploration into these topics.

The thesis is intended to apply to a broad range of software systems used by the military and Department of Defense. Despite the focus on UAVs, the presented methodology is designed so that parallels can be drawn and ideally applied to a wide range of software systems with varying objectives. As discussed, the primary context of the system analysis in this study is the unmanned aerial system context. This operational environment nominally involves one or more UAVs being controlled autonomously or via human-in-the-loop constructs, with communication and/or data links possibly providing information to external (typically ground-based) command stations. As these UAV systems rapidly continue to advance with the emergence of new technologies, the role of software and its integration becomes ever increasingly important for achieving successful system-of-systems operation.

Such holistic perspectives may further facilitate future concepts employing these UAV capabilities. One example of this type of system-of-systems is a UAV swarm cooperating to achieve an objective. Research is currently being conducted at Naval Postgraduate School (NPS) involving swarm concepts, which are of particular interest to operational stakeholders such as Pacific Fleet (PACFLT), Office of the Chief of Naval Operations (OPNAV), Office of Naval Research (ONR), United States Coast Guard (USCG), Defense Advanced Projects Research Agency (DARPA), and Marine Corps Warfighting Laboratory (MCWL). As such, this thesis is conducted in support of the Secretary of the Navy initiative for the Consortium for Robotics and Unmanned Systems Education and Research (CRUSER) and projects being conducted in the Advanced Robotic Systems Engineering Laboratory (ARSENL), which utilizes OSS extensively to demonstrate swarm capabilities.

1.3 Benefits of Study

This thesis highlights the potential ability of OSS to be embedded in DOD hardware systems by analyzing a case study of OSS programs used in UAV swarms. The analysis provides a methodology for the evaluation of OSS programs under consideration as alternatives to proprietary software, and provides tools to streamline data gathering from OSS repositories.

The thesis also bridges the gap between DOD software development community and the rapidly-growing OSS community by discussing recent trends in software development, including the influence of open source developers on robotics software and unmanned system software. It calls attention to these issues to help widen the field of view of DOD personnel as they consider future methods for development of software dependent systems. More specifically, the thesis benefits unmanned systems development in the Department of Defense by dispelling rumors of OSS inadequacy, and discussing real world examples of OSS programs successfully implemented in DOD mission environments.

Finally, the thesis is expected to also benefit the open source software development community by providing an additional tool for use in evaluating projects and potentially guiding their development in a manner that aligns with operational and/or acquisition requirements. For example, the idea of survivability is not discussed significantly in OSS literature, but it provides a context within which military programs may be evaluated. The process for survivability analysis discussed in the thesis provides an initial blueprint for evaluating software projects based on data generated from their source code repositories. The methods for defining individual event probabilities for attacks on open source projects could allow project managers to refine their efforts to build quality software and to strengthen their software development communities.

1.4 Research Questions

This thesis addresses several specific research questions. The questions are designed to provide some direction and scope for the research without overly constraining the analysis or limiting the resulting findings. The primary goals of this thesis are captured in the following research questions.

1. How can the concepts of traditional aircraft combat survivability be used to evaluate the capability of OSS to withstand an attack from an adversary?
2. What attributes of an OSS software program make it more capable to withstand an attack from an adversary?
3. How can OSS project data be retrieved in a repeatable manner in order to feed OSS survivability analyses?
4. How could an adversary destroy a system or force a mission failure by manipulating software embedded in the system?
5. What is the best way to retrieve publicly-available data from websites that host OSS project repositories?
6. How can OSS project data be analyzed and presented to provide decision makers guidance for selecting suitable OSS programs?

1.5 Methodology

This section discusses the methodology used to address the research questions proposed in Section 1.4. Tools used to answer the research questions include a literature review, data gathering, programming of software scripts for analysis support, and a survivability analysis. The information obtained throughout the research is combined and evaluated to provide results conclusions, and areas for future work, as discussed in Chapter 5.

1.5.1 Literature Review

A thorough review of literature related to OSS and DOD system software is conducted, including news articles, websites, books, journal articles, research papers, conference proceedings and working papers. The materials are surveyed and organized, such that the body of information and sources can be easily accessed as the research progresses. The initial review enables a foundation for relevant OSS topics and supports subsequent methodology development regarding OSS capabilities.

1.5.2 Data Gathering

In order to compare and contrast OSS projects, data is required to help analyze their qualities. Due to the open nature of OSS, most projects are hosted on websites that provide de-

scriptive information about, for example, the people who work on the software, the amount of work completed on a monthly basis, and the overall size of the software project (as measured by source lines of code (SLOC)). While readily available, such data must be retrieved and organized in a fashion that facilitates analysis of interest. Investigation and development of supporting software scripts helps with data collection and aggregation processes in support of efficient and reproducible results.

1.5.3 Analyses

The primary method for analysis of OSS programs is borrowed from traditional aircraft combat survivability. Methods for survivability analysis are adapted to fit the OSS development paradigm, including the identification of events that could allow an enemy to exploit the OSS development process by introducing vulnerabilities into software embedded in an autonomous UAV. A good survivability analysis is based on a thorough understanding of the events that lead to a kill and a good definition of the event probabilities. Several existing OSS programs are examined to determine values for individual event probabilities in the sequence an attacker might use manipulate a program.

Probability definitions are based on data obtained from projects' source code repositories. Most OSS is stored on web-based repositories that provide access to community members who work on the projects from different geographic locations, and due to the open nature of the projects, the data is often accessible by the general public. GitHub, a popular website that hosts many of the software programs used by ARSENL, provides most of the data incorporated in this thesis, and a software program is written to help fetch data from GitHub. Analysis of several OSS programs provides an indication of the programs' suitability for use in DOD systems.

1.6 Thesis Outline

The thesis is organized by chapters. There are a total of five chapters, beginning with an introduction and ending with results and conclusions. This section discusses what the reader can expect in each section.

Chapter 1 provides an introduction for the thesis. It discusses minimal background infor-

mation and outlines the scope and research questions to be addressed. It substantiates the subject matter and gives the reader an overview of what to expect from the analysis.

Chapter 2 provides a review of the literature related to the thesis topic. It discusses findings from journal articles and reports on OSS development and use. The chapter gives the reader sufficient background knowledge to comprehend analyses discussed in following chapters.

Chapter 3 discusses the survivability analysis that is performed. It gives an introduction to traditional survivability techniques and discusses the way they are adapted for use in this analysis. It provides a scenario for the open source development process to define how an adversary might attack a specific project. Data for several projects are examined to provide a means for quantifying their performance and capability of defense.

Chapter 4 details methods used to gather data about OSS projects used in the analysis. It discusses a software program that is developed to help retrieve data from GitHub, and presents the data that is gathered.

Chapter 5 discusses conclusions and results of the thesis. It provides recommendations for the use of open source software in DOD systems. Finally, it discusses possibilities for future work and topics that need to be examined further.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Literature Review

2.1 Introduction

Chapter 2 introduces topics related to the analysis covered by the thesis, including the software development process, open source software, DOD software, UAVs, and UAV software. Content in this chapter provides background information that supports subsequent discussion, and also acts as a reference guide for later chapters. Most of the material discussed in Chapter 2 is derived from various outside sources, which are provided at the end of the thesis in the reference list.

2.2 Software

Software is known as "the entire set of programs, procedures and routines associated with the operation of a computer system" [6]. It contrasts hardware, which is the physical components of a computer or system that are controlled by software. Software engineering was born when computers began to have a significant role in technology in the 1950s, but it was not until the late 1970s when software was available for development by hobbyists due to increased accessibility of personal computers. In its short lifespan, software has taken on many unique forms, its capability has grown immensely, and its presence in everyday life has become ubiquitous.

In the context of this study, the word "software" is used to describe the programs that cooperate with hardware to control or operate physical systems. There is some discussion of software as it exists on a desktop computer or network server, but the primary focus for software investigated in this thesis is in the context of military combat systems. An example of such software is that which is used to, e.g., manipulate the control surfaces on an autonomous UAV. This software can, for example, accept inputs such as stick positions or waypoint locations from a ground-based pilot (human or computer), and deliver output to hardware components such as motors or servos onboard the aircraft, which adjust their states based on the commands received.

This type of software is already used extensively in every branch of the military, and is often significantly large and complex, requiring substantial resources for its design, development, maintenance, testing, and deployment. Even more pressing, the necessity of such software is increasingly critical in DOD systems, further requiring matched growth in the ability to educate, train, and manage associated personnel.

2.2.1 Trends and Importance

Software has been a major part of technology development in the last 50 years, although its development and use has grown most substantially since the end of the 20th century and into the 21st century. The spectrum of its utility continues to expand in new and innovative directions. A measure of the extent of this expansion is to look at the demand for software engineers and developers. From 2007 to 2012, the number of software jobs in the U.S. grew by 31 percent, which reflects a rate that is three times faster than the overall job rate growth in the U.S. [7]. The increase in *available* job positions for programmers also indicates the demand signal for software is getting stronger. For example, U.S. automaker Ford Motor Company reports that their most essential positions involve software and systems engineering [8]. An examination of the modern automobile immediately highlights why Ford requires so many technically, and specifically software, oriented jobs, due to the significant integration of software into many automotive sub-systems. For example, computers are no longer limited to mechanical sensors and timing control, but are major elements used in environment control, entertainment systems, electronic components, engine performance monitoring, directional control, and enhanced braking. All of these computer-controlled components require software to run properly, and increasingly, major automakers must employ a team of software engineers who can design, build, integrate, and maintain this software.

This trend in software usage is paralleled in the DOD with increasingly complex and integrated combat systems. In order for the U.S. to remain relevant among world competitors, it must match and exceed other countries' military technology development. Just as in commercial products ranging from smart phones to home appliances to automobiles, software continues to play a substantial role in the upgrade and/or creation of nearly all combat system capabilities. Due to software's importance, the DOD has a strong interest in the way

it is procured, developed, secured, used, deployed, stored and maintained. New combat system programs are extremely dependent on advanced software to complete their mission requirements. The F-35 Joint Strike Fighter is one example of a sizable program that is burdened by rising software costs due to its high complexity. As recently as 2014, major testing has been delayed due to software development problems resulting from uncertainty in software requirements and software version control [9].

2.3 Open Source Software

Software produced today can be placed in one of two broad categories based on the way it is licensed, proprietary software and open source software. Proprietary software is written with the intent of maintaining control over the source code that lies behind the various programs, and companies that create proprietary software treat their work as if it is patented. They prevent users from understanding exactly how the programs work in an effort to maintain an edge over competitors. These types of programs typically come with sizable licensing fees that customers pay up front, and these fees can present a substantial cost, especially if multiple copies of the software are required.

The term "open source software" is used primarily to describe a software product that allows the user to access the source code behind it. The Chief Information Officer (CIO) for the DOD defines open source software as "software for which the human-readable source code is available for use, study, reuse, modification, enhancement, and redistribution by the users of that software. In other words, OSS is software for which the source code is 'open."' [5]. This type of software is controlled by special open source licenses, and there are several unique licenses which differentiate OSS projects by governing how source code can be manipulated and redistributed (see Section 2.3.2). The philosophy of open source software promotes sharing of ideas instead of protecting them, and it aims to achieve a higher level of technology by using cooperative techniques. It has been tremendously successful in certain applications such as the Mozilla Firefox web browser and the Apache web server [10].

Open source software is often mistaken to mean "free" software. It may be true in some instances that open source software is offered free-of-charge, but there are other instances when it is not. It is important to understand that using OSS will not always be cheaper

than using a proprietary equivalent. Often times the initial acquisition cost of open source software is small or nonexistent, but there may be costs tied to the software for things like training, software upgrades, and maintenance [11]. Cost is always a big driver of software selection, and it should be considered even when using OSS acquired free of charge.

2.3.1 Open Source Software in the DOD

Though perhaps not immediately apparent, the DOD, in fact, uses OSS in many ways. According to an assessment of various programs in the DOD conducted over a decade ago, the DOD was already using at least 115 open source software programs [12]. In 2003, the software was used in applications such as infrastructure support, software development, security, and research, and in recent years, the role of OSS has expanded to computing infrastructure, graphics, geospatial imaging, database management, and modeling and simulation [13].

Furthermore, the trend of increased OSS usage does not show signs of slowing. Figure 2.1 shows the estimated number of open source projects by year since 2007. The rising trend of the data in Figure 2.1 indicates increasing popularity of OSS, and when combined with DOD policy preferentially adopting open source solutions [5], such growth in OSS is likely to become more prevalent in years to come. As such, going forward, it is increasingly imperative to understand how to evaluate OSS in order to select well-suited projects, if available.

2.3.2 Open Source Software Licenses

The role of intellectual property is of significant relevance in any discussion on OSS, and copyright law provides protection for developers who write software code. Generally, source code cannot be adjusted, reused, or redistributed without the specific approval from the author. If developers want to allow users access to the code that they write, approval is facilitated through a software license bound to the code. Most proprietary software restricts users from seeing the source code and prohibits any adjustment or redistribution. Such restrictions are the primary distinctions in which OSS differs from proprietary software.

Licenses can be written to include essentially anything that the code's author desires. Pro-

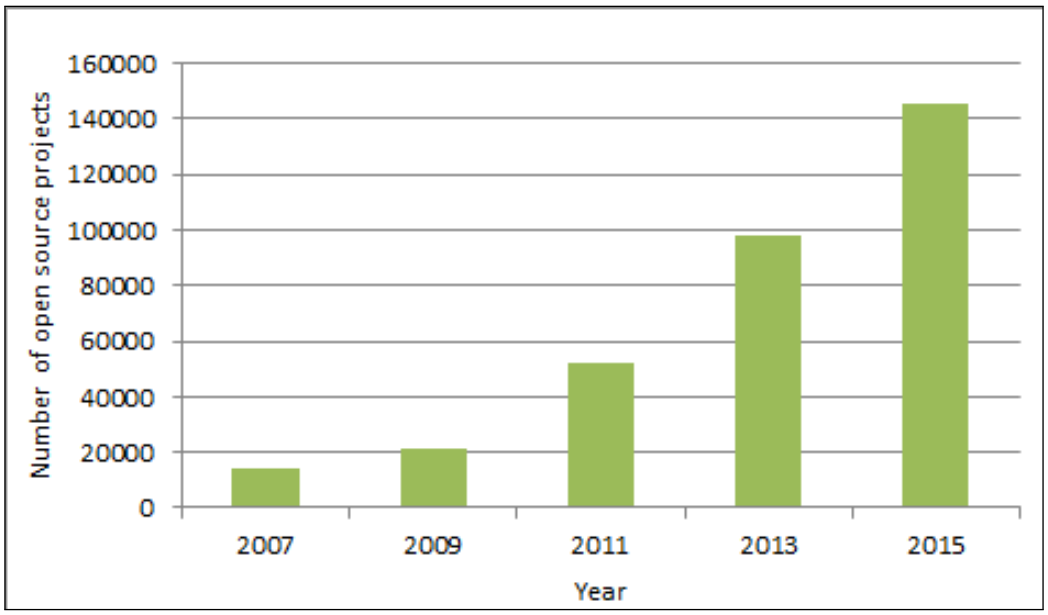

Figure 2.1: Number of open source projects over time since 2007. Adapted from [14]

grams under the umbrella of OSS typically share some general characteristics. Arguably, the most important quality of OSS is that it is termed "free." The word "free" is not used in the economic sense, but in the sense of liberty, as in the freedom to do what one wants with the product. Open source licenses aim to guard this freedom and protect the programmers who build open source code.

Developers have created "free" programs since the beginning of software to maximize creative development and to learn from one another. It was not until 1998 when a formalized definition of OSS was created. At this time, the Open Source Initiative (OSI) was established to provide governance over open source license definitions. OSI is a non-profit, community-recognized body for reviewing and approving licenses that conform to their definition of open source [15]. The following requirements must be included in an OSS license for OSI approval.

1. Must allow redistribution of the software
2. Must make the source code available with the program
3. Must allow modifications and derived works
4. May restrict source code distribution under certain circumstances

5. Must not discriminate against persons or groups
6. Must not discriminate against fields of endeavor
7. Must apply to all to whom the program is distributed
8. Must not be specific to a product
9. Must not restrict other software
10. Must be technology-neutral

The software community claims these attributes are essential for a program to be considered OSS. While there are several types of OSS licenses in existence, they all share these baseline characteristics.

The vast majority of operational open source software is governed by a relatively small number of licenses. These include the Apache License (2.0), the Massachusetts Institute of Technology (MIT) License, the GNU General Public License, and the Mozilla Public License 1.1. It is good to have a general understanding of the primary licenses that are used because 90 percent of all OSS software uses just the top 10 license types [16].

Awareness of the details of the governing license is essential for any software development initiative, whether open or otherwise, to avoid infringing on copyright law. For example, the GNU General Public License requires that modified code be redistributed under the same license. For a government project, such a requirement may preclude the use of software for certain use cases or applications, as, for example, the DOD may not want developed software to be redistributed to the public for security and/or classification reasons.

Perhaps even more relevant in the open source community, OSS licenses are an important part of software development, as they protect both the authors and the users of OSS programs, and additionally, give developers freedom to define how their code must be used.

2.3.3 Open Source Software Development Process

The development process for OSS differs depending on the project. Many software developers enjoy these types of projects due to the inherent freedom they provide. Although OSS development does not follow one single set of guidelines, most successful projects employ some sort of structure that mimics one or more of those used by companies developing proprietary software. One example is the Apache web server project, which is a widely used

open source software program designed for server computers for hosting websites. Now a large project that has been under development since early 1995 [17], the vast majority of websites use this software, and Apache is considered very secure and robust [18]. A previous study analyzed several aspects of this and other open source software programs [10]. In the case of Apache, it was noted that among the community of developers contributing to the project, there exists a well-defined structure and hierarchy for different types of developers. The developers who contribute the most and are active the longest typically have higher levels of responsibility and access to the primary source code.

Many projects' organizational structures are similar to Apache's, where often, developers can be grouped based on their level of involvement. Some examples of unofficial titles of people submitting code to the project include "release manager," "senior core group members," "core group members," and "standard developers." For example, Table 2.1 lists the developer titles used in the Apache program and their proprietary counterparts [10]. Despite the difference in the openness of the software, these positions within an OSS project hierarchy often reflect similar software engineering positions that a commercial software firm, such as Microsoft, would employ for a given software project.

Apache Project (OSS)	**Proprietary Software**
Release Manager	Project Manager
Senior Core Group Member	System Architect
Core Group Member	Project Member
Standard Developer	Tester

Table 2.1: Comparison of titles used for developers on the open source Apache Project and representative proprietary software projects

No matter how loosely structured a development process for OSS, successful programs can be seen to follow common development management schemes. Open source software is most often created by "massive parallel development" [19], in which, typically, a single lead developer (or group of "core" developers for a large project) organizes the project and guides its direction. Secondary developers write code, test the software, and can often make recommendations for the project.

In a study looking at several sources describing the OSS development process, Acuna notes that Institute of Electrical and Electronics Engineers (IEEE) standard practices are used for

many OSS projects [20]. IEEE defines five categories of design which encompass 28 total activities. Of the 28, 15 areas are specifically mentioned in the studies examining different open source software projects. Despite the popular theory that OSS development varies drastically from proprietary software, in reality, there are more parallels and similarities than there are differences.

Largely in response to potential misconceptions surrounding OSS, the DOD chief information officer published answers to several frequently asked questions regarding OSS [21]. The paper addressing the FAQs discusses how code progresses from primary developers to the finished product. The project is first created and the source code is placed into a trusted repository on the internet, where only certain developers (i.e., "trusted developers") have access. The trusted developers are typically those who either started the project itself or those members of the software development community hand-selected by the founding developers. Once the project is online, general users and developers typically have access to the source code and can download, manipulate, and distribute it as they wish. As they continue to use this software, these developers can often directly provide feedback in the form of bug reports and modification requests. Although some users can change the existing code in order to make improvements or address a bug they have found, these users cannot simply change the source code that exists in the trusted repository because they are intentionally not granted access. For a general developer to incorporate his changes into the original source code, the trusted developers first evaluate the potential contribution for quality and adherence to compatibility and coding conventions, and then allow the change if they approve said changes or additions. In this way, OSS can progress from a small project to a large repository of code that is potentially actively used, maintained, and updated by a large group of software developers.

One notable stark contrast to proprietary software development is the common lack of face-to-face meetings that occur among OSS software developers, and as such, OSS projects are usually characterized by large numbers of involved individuals who are not usually co-located (nor need they be). Such teams of OSS developers leverage remote communication and collaboration tools, such as online repositories and software management systems, real-time chat and discussion forums, email distribution lists, as well as conventional websites to maintain connectivity. The source code usually resides on a server typically readily

accessible via the internet. Using these techniques, developers remain aware of the ongoing progress of the project and the contributions of community members. Although this visibility and transparency is often touted as a strength of OSS, as will be seen in the analysis presented in this thesis, such access via online portals provides a potential avenue for anonymity among people working on the project, in that members can often join an open source repository and provide inputs without supplying their true name or other identifying information.

Developer Incentive for Contributing to OSS

At first glance, one might think it counter-intuitive that a software developer would want to share work freely after spending time and effort in the software's implementation. In other words, not immediately evident is how such a system could continue to exist, with developers producing code and distributing it without payment or exchange of goods in return. However, as detailed below, numerous motivating factors continue to inspire open source programmers and allow these types of projects to progress, let alone exist.

A possible reason an individual may be motivated to engage an open source software project is his or her interest in learning about how a particular program or function works. By joining an open source project, the developer gains access to a wealth of information provided by a diverse team of programmers of varying skills and backgrounds. One study highlights that "learning is one of the driving forces that motivate developers to get involved with OSS projects because it provides the intrinsic satisfaction for OSS developers" [22]. These learning opportunities hold value that cannot necessarily be measured in quantifiable dollars, but is measurable in education and increased technical abilities [23].

Other OSS contributors may be paid employees of commercial corporations assigned to work on open source projects, as it is in the interest of commercial software companies such as Google or Microsoft to provide inputs to open source projects. With OSS popularity increasing, large projects are beginning to have significant influence in the software community as a whole. These companies want to ensure that they remain aware of technology progression and provide direction at times. One of the ways they do this is by maintaining a paid staff dedicated to being involved with open source projects. One example is the Dronecode [24], which is an open source project structured under the Linux

Foundation. The project was founded and is still supported by commercial entities such as 3D Robotics, Baidu, Laser Navigation, and Walkera [25], which employ software developers to actively participate, guide, and/or lead development efforts.

Programmers can also use open source collaboration for résumé building. If a developer provides a significant impact to an open project, such investment in time and training has often been rewarded with additional employment opportunities or advancement. In addition to potential job offers arising from demonstrated software development skill sets, a current employee might also use open source work as a measure of professional development and value to negotiate a pay raise. Despite the general notion that programmers build OSS without selfish interest in mind, there are many ways to profit from such efforts [26].

Although monetary motives are plausible and often likely, some people enjoy writing code simply for the satisfaction gained by helping others. This altruistic mindset can have a lasting impact on the software community, as mentors teach apprentices and knowledge is passed freely among collaborators, as evidenced by the success of various online discussion boards providing such help forums, including those focused on computation and/or programming questions such as StackOverflow [27]. Many software developers share a common desire to help each other progress in their field [28], ultimately leading to not only progress of the software project, but also the cultivation of an increasing population of software developers that can contribute to that and other software projects.

2.3.4 Advantages and Disadvantages

Regardless of motivation, OSS continues to gain popularity across diverse swaths of application areas, research domains, and consumer sectors. In providing a perspective or philosophy that may be different from traditional software development methods, various benefits or penalties exist to the use of open source software.

The advantages of open source software are often debated due to the number of people and companies on either side of the argument, however, there is research that shows some undeniable benefits of existing open source software programs and their impact on the DOD. The benefits are technically oriented, such as higher security and higher reliability, and business oriented, such as lower cost and higher business functionality. Many of the

highlights presented here summarize a study on the benefits and drawbacks of OSS [29], in which the study team interviewed 13 software and information technology (IT) firms to draw from their experience with OSS.

Competition

Competition among vendors is generally accepted as being healthy for the software engineering market. Open source software allows people with similar interests and goals to come together and work toward a common cause. These people are often motivated by personal interest and often work on open source projects during their free time, and may build software because the commercial-off-the-shelf (COTS) software already in existence does not offer a solution that satisfies their needs. As these OSS initiatives develop and offer new software alternatives, commercial suppliers are forced to honor the upgrades and advances in features and functionality. Ultimately, the extra competition leads to better software technology. Even if OSS is not used directly, the existence of these options such as OpenOffice [30] and LibreOffice [31] forces commercial programs like Microsoft Word to stay current and keep up with competing programs. The competition also affects the cost of commercial products. Without open source products on the market, proprietary software would likely demand a higher value which would translate to higher cost for the consumer.

Security

There are some open source products designed specifically with security in mind. One such product used by the DOD is Open Berkeley Software Distribution (OpenBSD), an operating system that touts its security and reliability as one of its strongest features. Critics of open source software claim that due the accessibility of OSS, any foreign entity could infiltrate a critical program with malicious software [32]. Proponents defend open source software, saying that malicious software would have to pass under the watchful eye of every developer working on the project, which could number in the hundreds or thousands of programmers for large projects. Morgan and Finnegan's study highlights that the majority of firms interviewed thought OSS provided higher security due to "the availability of source code, the reduced threat of viruses, and the extra awareness of security in the design phase of projects." [29]

When considering the security of software, it is important to note that regardless of whether a piece of software is open source or not, there is potential for it to have security flaws. As such, numerous researchers have studied and generated statistics comparing the security flaws of proprietary software to open source versions (such as Internet Explorer compared to Mozilla Firefox), which typically conclude that vulnerabilities are, in fact, much more prevalent in commercial versions [33]. However, the implicit benefit leveraging a larger number of people working with OSS to improve prevention of bugs from being introduced or residing in the code base relies heavily on the skills and initiative of the specific user and developer community for that software project. For example, some open source projects are highly active and are likely to better exhibit the aforementioned advantage, whereas other projects are less actively managed or developed. In the latter case, if software users do not search for issues, the issues have a higher probability of remaining in the code [34]. Even though these studies find in favor of OSS in many cases, OSS under consideration for use or inclusion should be carefully assessed to include evaluation of the vibrancy of the given project as a potential indicator of its level of security.

Creativity

One measure of creativity is the speed at which functions are developed for a software project. This "growth rate" projects experience serves as an indicator of the level of innovation that a project possesses. Paulson *et al.* gathered empirical data showing creativity to be higher in open source projects when compared to proprietary projects [35]. Typically growth is highest at the start of projects and decreases over time. The study compared the growth rate of proprietary and open source projects by analyzing the number of functions created over the lifetime of the projects. The results consistently showed that the number of functions created, as measured by growth rate over their lifespan, was higher for open source projects.

There could be several reasons for higher creativity in OSS, including the intrinsic motivation for OSS programmers to help their fellow community members [28]. Also, good developers in open source communities are recognized and typically highlighted quickly, while commercial employees often cannot receive broad recognition (outside of the company) by releasing their code. Open source developers can select which software they develop, and tailor their work to applications that interest them most.

Further, usually people who write code for open source projects are users of the products. This vested interest spurs motivation to design high-quality code that meets their current and future needs. OSS also encourages users to view and be familiar with projects that are similar to the ones they are working on, which has the additional benefit of enhancing the community's understanding of the technology involved and allows developers to utilize portions that have already been accomplished. As such, availability of existing open source projects avoids having to reinvent sections or sub-routines or entire libraries. In this way, open source software can potentially support a faster development process and enable advancing the software technology quickly.

Progression

The fact that OSS is used by a growing number of people is a good indicator that it has significant advantages over proprietary software. Over 60 percent of corporations currently use OSS and consider OSS options before looking into proprietary programs. Over 50 percent of these companies also say that at least half of their engineers participate in open source projects [14]. Companies are clearly seeing good results from open source programs, as evidenced by their willingness to fund OSS projects and their adoption of OSS applications for their own use. Perhaps the biggest indicator that OSS is becoming more embedded in the software community is that companies are paying to be involved by allowing their employees to make open source contributions without receiving specific compensation.

Disadvantages

In the previously cited study, Morgan and Finnegan also highlight several disadvantages that companies have experienced using OSS [29]. One of the most common is lack of expertise, which often prevents people from considering open source options. Often, project managers fear that their employees have too little experience with the OSS to facilitate a smooth transition. The apprehension can often be attributed to unfamiliarity, which could potentially be overcome by introducing open source concepts and including such OSS options when considering types of software to use, purchase, or integrate.

Another common disadvantage is often the lack of documentation [36], which is consis-

tently a larger problem for OSS than for commercial projects. The root of this problem for certain OSS projects may be due to the given open source process being less formalized than in proprietary companies, or due to other reasons such as the absence of self-regulation or governance over development procedures for open source, or simply less time devoted to proper documentation. Another issue is that the primary users of OSS are often the developers themselves, and do not need as much documentation as an end user who did not participate in building the code. As a result, novice-friendly documentation may be harder to generate. In contrast, the end user of commercial products is not only unfamiliar with the development, but cannot view the source code, so the documentation must necessarily be focused on clarity and simplicity.

On the business end, users of OSS may often find it difficult to adopt the software because there is a lack of support. Despite acquiring an application free of charge, support often comes at a premium and can sometimes be inadequate or slow. Another problem for businesses is often the lack of accountability in some open source programs, and companies may have liability concerns when using OSS. OSS licenses almost always contain a clause indicating the authors are not responsible for anything the user does with the code. In contrast, proprietary companies can have reliability or security guarantees written into contracts to protect the customers.

2.4 Unmanned Aerial Vehicles

As the prevalence of UAVs increases, so to does the software enabling automated or even autonomous capabilities in these systems, as the commands performed by a pilot in a conventional aircraft are replaced by software-generated commands in an UAV. To highlight this point, the U.S. Air Force began developing unmanned aircraft in 1960 after concerns over pilot attrition during the Cold War [37]. Since then, the drone market has expanded in the military and in civilian sectors substantially. In 2010, 41 percent of aircraft in the DOD were unmanned [38]. Figure 2.2 shows the expected increase in market capitalization for drones in civilian and military sectors [39].

Two types of UAVs exist. These are remotely piloted aircraft and autonomous aircraft. Most UAVs used in current combat are remotely piloted, meaning a person makes nearly all decisions related to basic flight and mission. Other aircraft are autonomously controlled,

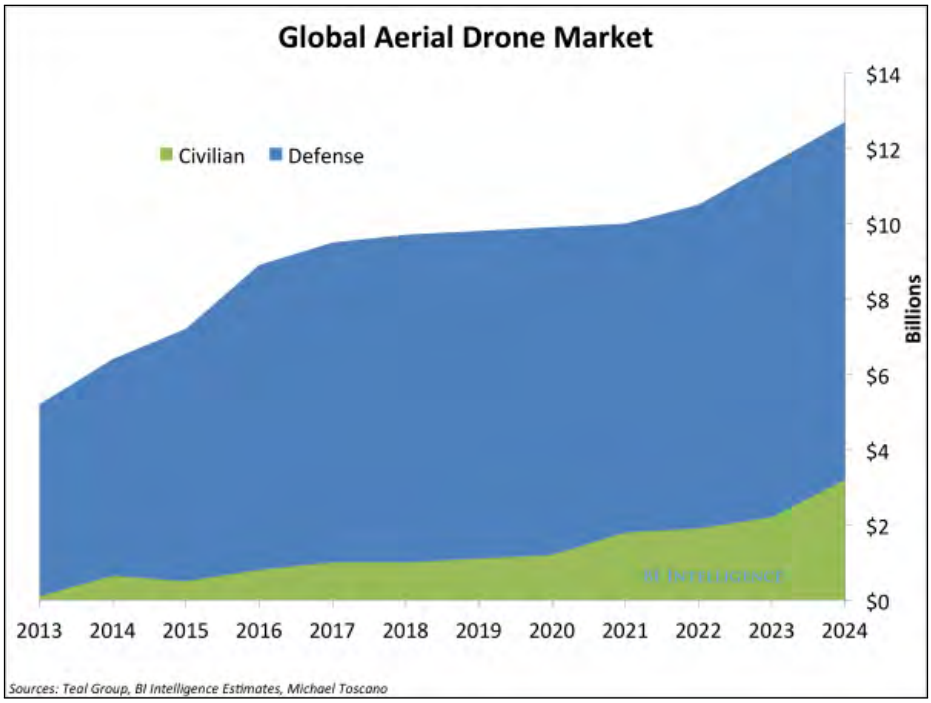

Figure 2.2: Global aerial drone market through year 2024. From [39]

meaning an aircraft operates without the intervention of a pilot [40]. A relatively new advent in unmanned flight is the use of multiple UAVs cooperating toward a common goal, also known as a swarm of UAVs.

2.4.1 Swarm Concept

Many of the OSS programs studied in this analysis are used by ARSENL as a part of the Aerial Combat Swarms (ACS) project. A group of aircraft is considered a UAV swarm if they can communicate with each other and perform cooperative tasks as a group [41]. Swarms can be advantageous due to the low cost of individual aircraft, and in the case of warfighting effectiveness, a group of several or many aircraft may have the ability to overwhelm advanced defensive systems that could otherwise easily handle one larger plane.

Live-fly (i.e., outdoor) swarm UAV research and development is currently limited due to civilian regulations prohibiting operations of two or more drones (another term for UAV) at once [42]. Some exceptions exist, leveraging access to military resources and venues, allowing for efforts in UAV swarms at various institutions such as NPS. For example, NPS

researchers have a goal to conduct experiments involving up to 100 UAVs in simultaneous flight operations [43] in the ACS project. Due to the numbers of aircraft involved in the operation, human interaction with individual aircraft is necessarily limited, and instead, success of the swarm will be dependent on software-derived capabilities via advanced autonomous software.

2.4.2 Software Components in the Swarm Environment

ARSENL uses software in several capacities during UAV swarm operations. On the individual aircraft, software is used in flight control systems, in payload management systems, in communications systems, in sensor processing, and in Command and Control (C2) systems. There is often a command station which utilizes software to provide individual and/or group instructions to various aircraft in the swarm. Depending on the size of the swarm, multiple stations can be used.

The State of Open Source Software for UAVs

OSS for UAVs has been in production since at least 2003 [44], as there are many projects designed specifically for drones including APM/Ardupilot, OpenPilot, and PaparazziUAV. The community is made up of primarily hobbyists who build software and hardware for small aircraft on their free time. OSS for UAV has recently drawn interest from large organizations such as the Linux Foundation, which, in 2014, enabled the Dronecode Project, which aims to unite software developers, hobbyists, and commercial companies in the common goal of creating a standard for drone OSS development. The project has already drawn 1200 programmers who provide over 150 daily code commits (submissions of written code) on some projects [25]. This support and popularity showcases the growing potential of open source solutions for drone software.

A partial list of software used by ARSENL at NPS in their software design and development in support of UAV swarm research is shown in Table 2.2. Licenses used by each program are shown for general reference, and more information regarding the differences between each license can be found online [45]. The association column indicates whether the OSS program is used directly for ACS development, or indirectly as a dependency for another part of the ACS development process. Many of the programs are related to one

Project name	Project function	License	Association
ArduPilot	aircraft autopilot	GNU GPL 3	ACS
MissionPlanner	mission control	GNU GPL	ACS
MAVLink	message marshaling	LGPL	ACS
MAVProxy	ground control station	GNU GPL 3	ACS
JSBSim	flight dynamics model	LGPLv2	ACS
Sik Firmware	firmware for radios	M. Smith	ACS
ACS ROS	autonomous functionality	(none)	ACS
PX4 Firmware	PX4 FMU driver	BSD 3-clause	ACS
PX4 Nuttx	real time OS	BSD	ACS
matplotlib	graph plotter	BSD, PSF	dependency
NumPy	scientific computing	BSD	dependency
eSpeak	speech synthesizer	GPL 3.0+	dependency
wxPython	GUI applications	WXwindows-3.0	dependency
wxWidgets	cross-platform GUI library	WXwindows-3.0	dependency
ccache	compiler cache	GPL 3.0+	dependency
NuttX	real time embedded OS	BSD 3-clause	dependency
OpenCV	computer vision	BSD	dependency
PIL	image processing	BSD 3-clause	dependency
autoconf	compiler	GPL 3.0+	dependency
arduino	electronics prototyping	GPL 2.0+	dependency
urllib3	connection pooling	MIT	dependency
CMake	build system	BSD 3-clause	dependency

Table 2.2: Open source programs used by the Aerial Combat Swarms project

another and they have several uses, including software development, aircraft control, aircraft communication, ground control operation, sensor processing, and data management. The majority of software used by the lab for ACS is open source. ACS is one example that showcases the potential of OSS. The programs that ARSENL use continue to provide advanced capabilities for conducting experiments with large groups of unmanned aircraft.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Survivability Analysis

3.1 Introduction

This chapter introduces the concept of survivability, explores its utility in the context of UAVs, and uses it to analyze the ability of the open source development process to defend against attack. The methodology for completing a survivability analysis is shown using data gathered from one OSS program, and conclusions are drawn regarding the program's resilience against such an attack.

3.2 Need for Survivability Analysis

Since the early 1970s, one of the most important fields of study for the development of U.S. military aircraft has been aircraft combat survivability. During the conflict in South East Asia, the U.S. experienced significant aircraft losses as a result of enemy integrated air defense systems, and in 11 years of conflict, approximately 5000 helicopters and fixed-wing aircraft were downed by the North Vietnamese [46]. By 1965, enemy air defenses were significantly developed since previous major conflicts including World War II (WWII) and the Korean War. The North Vietnamese's integrated air defense system (IADS) was more advanced, incorporating surface-to-air missiles (SAMs) acquired from Russia that were capable of reaching altitudes higher than the operational ceiling of most fixed-wing aircraft in the U.S. inventory. This forced aircraft into low-altitude attack profiles which were well inside the engagement envelope of anti-aircraft artillery (AAA) systems. Aircraft designers did not anticipate this kind of resistance and consequently, the aircraft were not well-equipped to avoid or sustain hits from SAM and AAA systems.

The tactics and combat systems employed by the U.S. were also inadequate to carry out the missions required during the conflict. The first laser-guided bomb was not fully developed until midway through the war in 1968, and up until this time, aircraft were using primarily unguided munitions (dumb-bombs) to destroy targets [47]. Between 1965 and 1968, Operation Rolling Thunder pitted the vast might of America's military power against supply

lines and strategic targets in North Vietnam. The primary goals of the operation were to prove that the U.S. was in control of the conflict and to destabilize the Democratic Republic of Vietnam. Unfortunately, after 643,000 tons of bombs were dropped and an estimated $ 900 million spent, the desired effects on the North Vietnamese government had not been achieved, and only $ 300 million in damage was estimated to be done. In addition, 900 U.S. aircraft were lost by the time the operation was halted. The aircraft were not designed to survive this type of environment, and the mission was not successful.

The concept of combat survivability was developed to understand why the U.S. was losing so many aircraft to enemy air defense systems. The DOD wanted to preserve planes and helicopters by making them less vulnerable and less susceptible to enemy fire. While originally intended for aircraft, the topic now impacts the design and development of many military systems, including ships and ground vehicles. Although aircraft combat survivability analysis has spread from aircraft to other vehicles and equipment, it is still primarily focused on the structure, hardware, and physical components of systems. Live-Fire Test and Evaluation (LFTE) is a statutory requirement for military hardware systems to achieve before they move beyond low rate initial production, but the testing primarily involves evaluating the effects of munitions and missiles on systems [48]. The following analysis extends the concepts of traditional survivability to software, applying survivability methods to evaluate how software embedded in a hardware system (such as an aircraft's flight control software) would survive an attack. Part of a survivability analysis involves identifying the various threats that a system could face during operation so that the system's survivability can be evaluated according to each threat. Section 3.3 discusses some of the threats faced by UAV software.

3.3 Threats

Due to the variable nature of factors that affect a system's ability to survive an attack, aircraft and ship survivability is always evaluated with a specific threat in mind. Survivability only pertains to one specific threat engaging one specific aircraft or system due to wide ranging threat capabilities and circumstantial aspects such as environmentals, tactics, doctrine, and operator quality that effect it. A UAV, especially one operating autonomously, relies heavily on software so that it can complete objectives without continuous inputs from

the human operator. UAV software could face threats from several sources, and the threats could target several portions of the software. In the following sections, threats are broken into three categories, unintended failures, unintended vulnerabilities, and malicious software insertions, and failure modes of UAVs are examined at the end of Section 3.3.

3.3.1 Unintended Failure

An unintentional failure could occur due to a simple flaw in the aircraft's software. Just like mechanical systems onboard a UAV, not all software always operates as intended by the designer, and there are many historical examples of software causing catastrophic failure and complete system destruction due bugs in the code. In 1996, the Ariane 5 rocket flew for approximately 40 seconds after launch before it self-destructed as a result of a software error. Figure 3.1 shows an image of the rocket exploding shortly after launch.

Figure 3.1: Ariane 5 rocket destruction after launch

This accident was attributed in part to software that attempted to convert a 64 bit floating number into a 16-bit signed integer so it could be used for a calculation [49]. The integer value did not capture enough information for the calculation due to its relative small size, and this caused an overflow in the computer hardware which led to loss of guidance and attitude information. With no guidance, the rocket quickly diverged from its intended course and its computer system initiated a self-destruct command.

A bug such as this could cause an unintentional failure mode in a UAV, and in order to prevent such a loss, thorough testing is conducted at the developmental and operational level to ensure that all modes of software operation are checked in anticipated operating

environments. The probability of this type of error happening is directly related to the talent of the software engineers who design the software and the technology involved in the software's code.

3.3.2 Unintended Vulnerability

Another possible threat to the aircraft's software is a vulnerability an enemy could take advantage of. All software has bugs (or vulnerabilities) [50], and sometimes hackers are able to exploit these bugs for their own good. This threat is similar to an unintentional failure, but it involves a bug that opens the software up to enemies, allowing them to control components of the system via the software vulnerability. This threat represents an enemy's accessibility to the rest of the system, and as such, these bugs could lead to the destruction of the aircraft or failure of its mission.

An excellent recent example of this from OSS is the "Heartbleed" problem [51] which enabled collection of encrypted data from secure web-sites. It should be noted that it was found by a paid Google staffer specifically looking for OSS vulnerabilities. This clearly makes the case for having DOD staffers employed in a similar capacity.

3.3.3 Malicious Software Insertion

The third and final threat to software is an intentional insertion of malicious code into the source of a program. In OSS programs, an adversary could attempt this by gaining access to the general development community involved in designing software for the UAV. Open source projects generally have limited restrictions for people attempting to join their development groups, so a malicious code insertion to a program in this setting is feasible. Whether or not the insertion remains with the source code long enough to make it into the program's release depends on the quality of the open source development process.

This type of threat could also occur in a program developed by a proprietary company. An adversary could gain access to the development community, potentially by being hired to work for the company, and once employed, he could insert malicious software just like in the case of an OSS project. Another way this could be accomplished is by hacking the proprietary software company's computer servers through the internet. Once the adversary

has a connection to the location of the code, he could adjust it or contribute malicious code.

3.3.4 Failure Modes

The mission of a UAV could be compromised by the failures in several ways. An aircraft could simply be diverted from its planned route profile due to initiation of altered code at a specific time after it began operations. The malicious software could do this by forcing the system to adjust pre-programmed track points and to disregard inputs generated from a friendly ground control station. The software could be programmed to land the aircraft at a location inside the enemy's country so they could capture the vehicle.

The software could also disrupt the sensors onboard the UAV so that they fail to record data, or spoof the system by providing false recordings. Depending on how it is designed, the software could operate in this way without the user knowing it, in which case the UAV could continue operations until the user realized they were getting bad information, effectively disrupting the mission as long as the code remained hidden. Ordinance control software could also be adjusted to render kinetic weapons useless, or worse, detonate them while they are still attached to the aircraft.

The prevalence of software embedded in the components of modern systems provides many avenues for failures to take place. This analysis specifically examines the survivability of an OSS project that faces the threat of a malicious software insertion, and the primary piece of the process that is analyzed is the OSS development cycle. In order to provide a frame of reference for the discussion, a scenario is presented in Section 3.4 which describes the type of attack being considered and the individual events that form the attack.

3.4 Scenario

The operational scenario in this analysis involves a UAV, an operator working through a ground control station (GCS), an adversary, and the flight control software required by the UAV system. For simplicity, only one software component onboard the UAV (flight control software) is examined in this scenario. The program, called FlightGear' is an actual OSS program that can be used as a flight simulator. Theoretically, an adversary could attempt to exploit others such as mission control software, navigation software, or communications

software, and interactions between these different software systems would likely affect the overall probability of survival of the entire system. Follow on research could include considerations for the relationships between programs and how the integration of the software onboard an UAV could affect the overall survivability of the aircraft.

Several key subjects are established to define characters and roles that are played in the scenario.

- **Trusted developer (aka "caretaker")** – The primary developer on a project, with authority to approve code submissions provided by the general development community.
- **General Developer** – A programmer who provides input to the project, including bug reports and newly designed code. A general developer does not have authority to merge his or her code with the original source. This is accomplished by the trusted developer
- **Adversary** – The individual whose goal is to undermine the software by creating malicious code and submitting it to be added to the source. He disguises the malicious code so that it appears harmless, hoping the trusted developers do not notice and accept it
- **User** – A customer who downloads the software and uses it for its intended purpose
- **Flight Control Software** – The software program employed by the UAV to control its position in three-dimensional space. For data gathering, this fictional software is represented by an actual flight simulation software used by numerous aviation- and UAV-related development efforts called "FlightGear"

3.4.1 Detailed Events

The operational environment involves the U.S. conducting sanctioned UAV operations near the border of an enemy country. On a periodic basis, the U.S. conducts intelligence surveillance and reconnaissance (ISR) operations across the enemy country border using low-observable stealth UAVs. The enemy country learns of the UAV operations inside its border and decides to take action, employing their intelligence team to retrieve information about the aircraft's characteristics including the software.

The enemy country decides to utilize a software programmer to attack the software used by the UAV (represented by FlightGear). The adversary examines the different types of software onboard the UAV and decides to attempt to alter the flight control software by gaining access to the user development community involved in constructing the OSS used for controlling the flight of the UAV. He then downloads the source code and explores it to discover vulnerabilities and to determine a method to exploit its underlying framework. He writes one or more pieces of code and requests the code be added to the source by a trusted developer.

The specific intent of the software code that he inserts is to input weaknesses into the program that could then be taken advantage of while the system is operational. This malicious code is then reviewed by the trusted developers, mistakenly accepted, and unfortunately merged with the source code, so that general users now have access to the new source code with the malicious software embedded in it. The adversary's code remains integrated with the source code until it is released and distributed to the public, at which point the user downloads the software package and checks the source code for errors and vulnerabilities, but does not notice the malicious code.

Believing the code to be safe, the user launches the UAV and initiates an autonomous mission. The aircraft automatically heads toward the enemy country border, crosses it, and begins flying a surveillance pattern and gathering information. The malicious software initiates, and the ground control station immediately loses contact with the UAV. The adversary's addition to the flight control program causes the aircraft to break normal protocol and fly into the ground, nullifying the UAVs mission and allowing the enemy to achieve an attrition kill on the aircraft.

3.5 Analysis

After describing the scenario, the survivability of FlightGear can be determined by analyzing the program. The approach used in this analysis involves breaking down the adversary's engagement of the OSS development process into parts, gathering and analyzing data related to the open source project, assigning probabilities to individual events in the engagement, and calculating an overall probability of survival for the scenario.

3.5.1 Essential Events and Elements Analysis

The first task is to isolate and define events in the engagement essential to the end result, which is a UAV no longer capable of carrying out its mission. These critical events can be determined by performing an Essential Events and Elements Analysis (E3A), which defines the events by examining the processes that lead to them. "Any combat operation, action, or incident that consists of a timewise sequence or combination of interrelated actions involving one or more parameters and variables associated with a system is referred to as a process [52]." Processes generally have one or more outcomes, referred to as events, which have associated probabilities they will or will not occur. The first step in defining the events is to define the processes by which they are created, followed by assignment of probabilities to the outcomes of those events based on data and analysis of the scenario. In addition to defining the events, questions are posed to help determine the probability of the events taking place in the given scenario. The E3A was conducted for the scenario presented in Section 3.4, and the results are presented in Table 3.1. The table begins with the final undesired event and proceeds in a reverse time line to end up at final entry, which is at the first undesired event in the scenario.

The next step in the process is to define event probabilities for each of the processes laid out in the E3A, which occurs in traditional survivability by gathering performance data about systems (such as an aircraft and an anti-aicraft SAMs) involved in the scenario. As such, in order for this analysis to be completed, a representative OSS program needed to be used as an example for the scenario. The program selected for this analysis was "FlightGear," and the data gathering process is explained further in Section 3.5.2

3.5.2 Data

The actual program called "FlightGear" does perform flight control functions, as the software in the scenario suggests, although it is is not a full blown autopilot, but a flight simulator that has similar functionality. Instead of having to control an actual aircraft, it controls a simulated object, but generally speaking, the purpose of the open source software does not significantly impact the outcome of the survivability analysis, and therefore it was selected to represent the fictional software in the scenario. FlightGear is selected in part due to the nature of the data available for the project and the format it is stored in. FlightGear is hosted

Essential Events and Elements	Questions
1. Friendly control of the aircraft is lost	What is the profile of the new flight path? Can control be reacquired?
2. Malicious code initiates programmed flight path	What is the intent of the malicious code? What impact does initiation have on the UAV's behavior?
3. Code remains un-flagged by every user group	How much time transpired between the malicious code commit and the UAV flight?
4. UAV user does not spot bad segment of code	How thorough was the user's search for anomalies in the source code?
5. Bad code passes final checks by trusted developers and is included in the final build of the software	Have the primary developers had previous success locating vulnerabilities in the source code?
6. Malicious code is not spotted by general developers working on the project	Is the general development community active? Have the general developers had previous success locating vulnerabilities in the source code?
7. Bad code is merged with the source	Who was the trusted developer that approved the merge? How did the adversary package the code to disguise its malicious intent?
8. Adversary provides commit including malicious code	How long was the user in the community before delivering the code?
9. Adversary gains access to the development community working on the UAV's flight control software	Are there any restrictions to joining the development community?

Table 3.1: Essential events and elements for the open source software development process

on a website called Gitorious.org [53] where the source code repository is also located, and from here, trusted and general software developers work on the code and submit updates to the source. Data regarding bugs in the project is stored on another website managed by Google [54], and is presented in tables which allow the data to be exported directly to a comma-separated values (CSV) file. Other data such as the number of active developers and the number of commits for the project was gathered from a website called OpenHub, which tracks information on many OSS projects [55].

After pulling the data from these websites, it was combined into a spreadsheet so that it

could be compared and analyzed. The different aspects of the program were compared and graphed to obtain trends in the data that indicated information about the health of the project.

Issues

The first metric analyzed was the activity level of the project, which was evaluated using the number of issues that were open and closed at a given time. In open source software, issues represent bugs or change requests added by general developers and testers, and they indicate the activity level of developers in the community working on the project. The importance of issues is further explained in Section 4.2.2. One of the ways that open source repositories allow projects to track activity is by including time stamps on issues that track when they are opened, when they are updated, and when they are closed. This data can provide a visual picture (such as the one shown in Figure 3.2) of the activity level of developers working on the project, showing how often issues are brought up by developers, and more importantly, how often the issues are solved and closed. One way to identify general trends in the data is to add trend lines, which is shown for FlightGear in Figure 3.3.

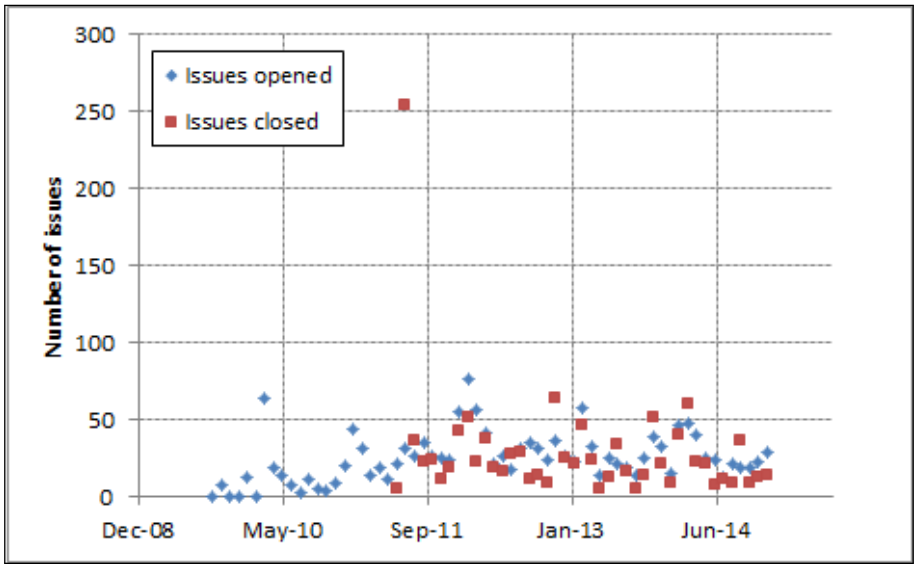

Figure 3.2: Number of issues opened and closed over time for FlightGear.

As shown in Figure 3.3, the general trend of issues opening during the time period was

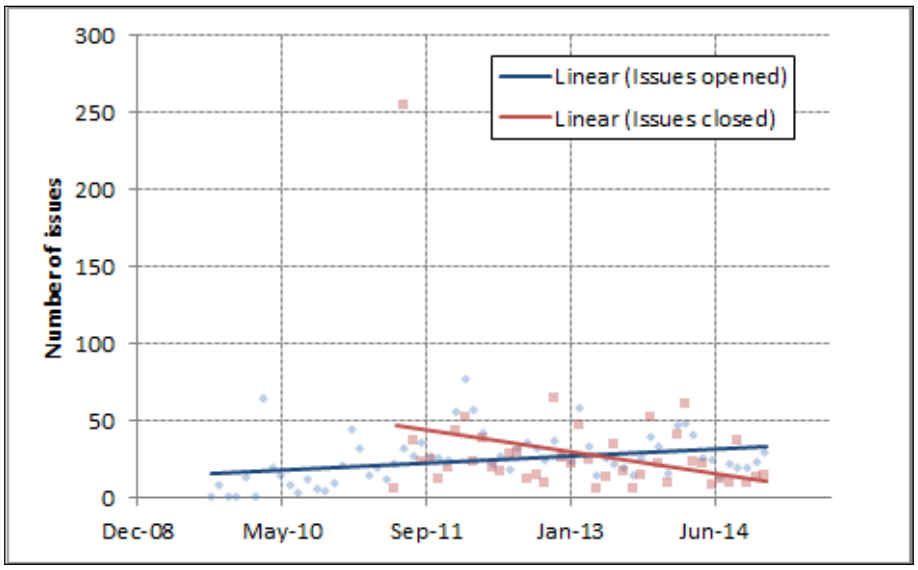

Figure 3.3: Trend lines representing the number of issues opened and closed over time for FlightGear.

positive, while the general trend of issues closing during the time period was negative. This was not a good sign for the project, as the scenario inevitably led to a large number of issues that remain open over time. This problem is illustrated in Figure 3.4, which shows the cumulative number of issues in an open state as time progressed during the analysis period.

As shown in Figure 3.4, FlightGear had a backlog which generally increased over time from July 2009 to December 2014. This indicates that there was a lot of work for the project managers to handle. As the software program approached release dates, the managers were forced to address the entire backlog of issues, which likely prevented them them from spending a lot of time on other important release tasks such as testing. This type of backlog increases the probability that an adversary's malicious code will pass incremental reviews and make it into the release.

Developers

Another method for determining the general health of an OSS project is by looking at the number of developers involved with the project, which provides an indication of general interest in the project and how many people are viewing the code. It is useful to compare

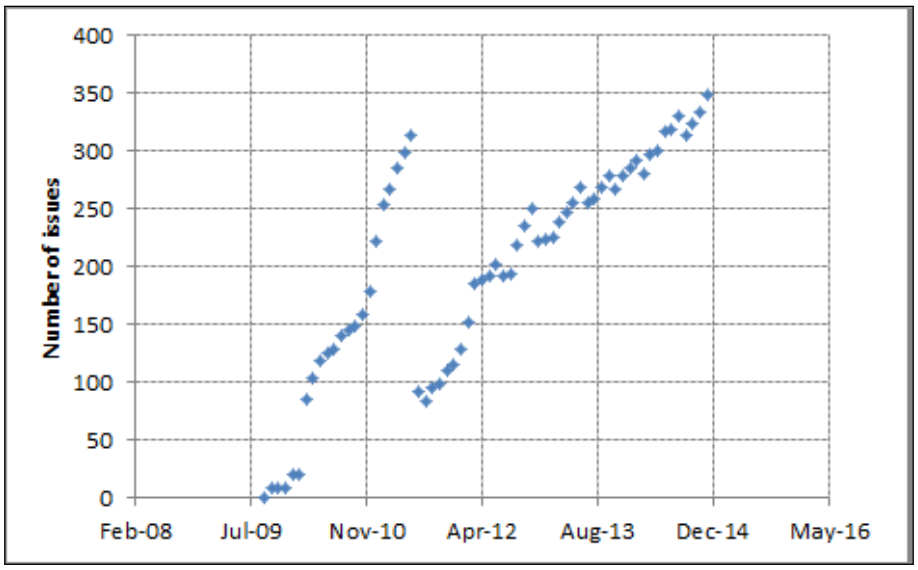

Figure 3.4: Cumulative issues, or backlog of issues, for FlightGear over time

the number of developers on a project to issues opened over time as a method for relating the different characteristics and determining justification for assigning activity levels to a project. Figure 3.5 shows the number of developers who were involved in the FlightGear over time and the number of issues that were opened over time.

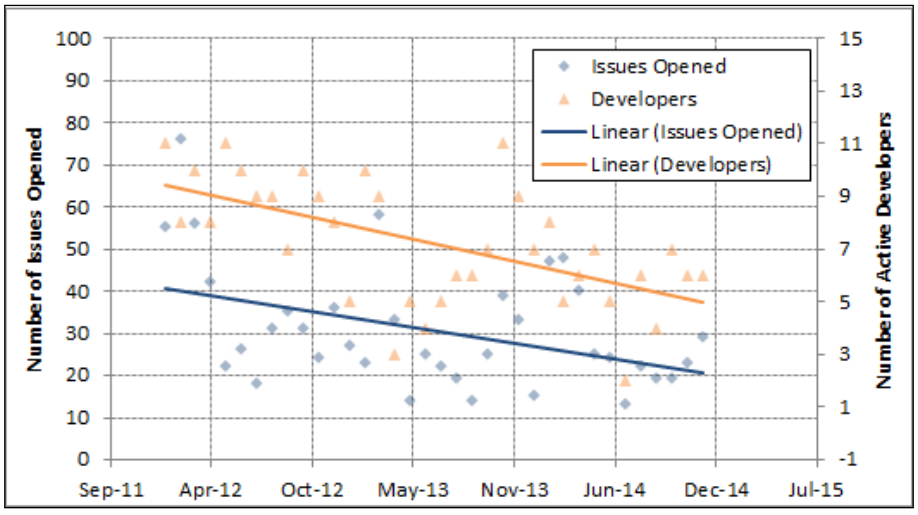

Figure 3.5: Number of developers who were working on FlightGear as compared to the number of issues generated for FlightGear over time

As shown in Figure 3.5, the number of developers working on the project decreased over

time as did the number of issues being opened. It is interesting to note that in the time sample shown in Figure 3.2 the general trend for issues opened during the time period increases, while in Figure 3.5 the issues opened shows a decreasing trend. This is due to the different time periods that are analyzed. In Figure 3.2, the time period runs from 2009 to 2015, while in Figure 3.5 the period analyzed is from 2012 to 2015. This indicates that toward the end of both periods, the project saw a significant decrease in developers, and consequently, activity on the project. The decreasing trend toward the end of the time frame could have been due to decreasing popularity of FlightGear, competing products, or mounting issues that developers were unable to control. Regardless, Figure 3.5 clearly indicates that the issues opened per month and the number of developers working on the project per month experienced a negative trend, which translates to a lack of health and a higher probability for an attacker to be successful in manipulating the software by passing unwanted code to the source.

Figure 3.6 shows similar data, but instead of the number of issues being opened, the number of commits each month is showed over time along with the number of developers working on the project. In open source development, programmers submit code for addition to the source repository in the form of commits, which generally arrive as text files that incorporate lines of software code. Projects with high numbers of developers increase their chance of creating quality software products, and similarly, high commit numbers in projects reflect high activity and good health.

Similar to Figure 3.5, Figure 3.6 shows decreasing levels for both developers and commits, which is not a healthy sign for the project. A decreasing activity level on an open source software program will lead to high workload for the developers who remain with the project and reduce the time they are able to put into each code commit and open issue. FlightGear's decreasing activity level decreased the programmer's capability of finding and weeding out malicious code presented by an adversary. This then increased the susceptibility of the software.

FlightGear's data was used so support hypotheses for individual probabilities that make up the survivability equation presented in Equation 3.1. Other considerations were taken into account and several assumptions were made to assist in defining the probabilities for the scenario described in Section 3.4. The next section discusses the probabilities for each

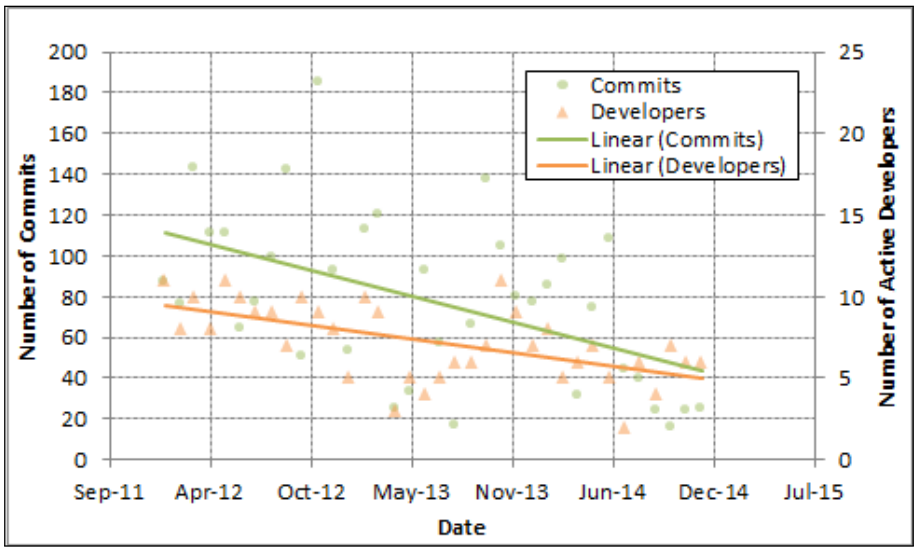

Figure 3.6: Number of developers who were working on FlightGear as compared to the number of commits provided by developers over time

event and assignment of probabilities based on the data gathered from FlightGear's project repository website.

3.5.3 Event Probabilities

The overall survivability of an open source software program is calculated using the likelihood of individual events that make up the process. There are nine events that contribute to an aircraft kill as outlined in Table 3.1, and they are listed here with amplifying descriptions and specific values for the FlightGear OSS project.

Probability of Access

The probability that the adversary will gain access to the general development associated with a given software project is based on the project's restrictions and the adversary's programming capability. Most OSS projects have very few restrictions preventing people from gaining access to the source code and providing contributions. The nature of OSS is one that thrives on inclusion and it encourages maximum participation to fuel creativity and cooperation in development. When restrictions are imposed on the development community, it limits the number of programmers involved and decreases the potential of the project, so

most projects require only a username, password and email address to join. While a developer's profile is viewable by the community, it is not often required to display identifiable information about the individual who created the account.

Not all projects are unrestricted, for example, the DOD hosts a website called www.forge.mil that requires a Common Access Card (CAC) for access [56]. This limits participation to members of the military, government civilians and employees of the DOD, which drastically decreases the probability of an adversary accessing the development community, but also limits the creative power of projects hosted on the website, as discussed above.

FlightGear is (formerly) hosted on a website called www.gitorious.org [53]. The website does not require any identifiable information to establish an account and an adversary would have no problem joining the development community, so the probability for him to gain access is 100 percent.

$$P_a = 1.0$$

Probability of Commit

The probability that the adversary would provide a commit to the source code is based on his ability to produce the code. Given the correct resources, a talented developer who knows the programming language could easily build a piece of code and submit it if given sufficient time and resources to build it.

For FlightGear, it was assumed that the adversary was well-versed in the programming language and the sophistication for the required code was low. This provided a high probability (98 percent) that he could provide a commit.

$$P_{c|a} = 0.98$$

Probability of Merge

The probability of merge is the first probability in the process that is impacted by the quality of the OSS development process in question. In order for the code provided by the adversary to be incorporated into the source code, it first needs to pass an inspection

by one of the trusted developers working on the projects. In this step, trusted developers look specifically for bugs and vulnerabilities inside the commit, and if they find something suspicious, they block it from the source code and likely terminate the user's presence in the development community. Not only does the adversary need to produce something that disrupts the aircraft's mission, he has to disguise it to appear as if it has another purpose. The adversary's ability to do this is largely dependent on his coding abilities.

The trusted developers cannot spend excessive time reviewing each commit or the project will not progress. If there is a large number of commits in the queue to be reviewed, they will not be able to allocate a lot of time to each commit and the probability of merge will increase. For FlightGear, the project backlog was high during the time frame under consideration, as shown in Figure 3.4. Due to the high backlog, the chance for the adversary to achieve a merge was determined to be higher than a typical OSS project would permit. Talented caretakers working with a low backlog could reduce the probability significantly, but for FlightGear it was determined to be 90 percent.

$$P_{m|c} = 0.90$$

Probability for Code to Stay

The next step in the development process is to wait for the source code to be finalized for a release. During the wait period, the general users have access to the code commit that was provided by the adversary because it is now part of the source. They can view, test and provide changes to the code as they see fit. In order for the adversary's code to end up in the final release, it has to survive the scrutiny of the general developers. The probability that it will stay is based on how much time transpires between the commit and the release, and the capability of the general developers to spot the malicious code.

For flight gear, two months was assumed to transpire before the final release, and the capability of the general developers was assessed based on the backlog of issues and the activity level. Activity level was evaluated to be average but declining as discussed in Sections 3.5.2 and 3.5.2. The backlog was unusually high, causing the developer's capability to be assessed as low, and the time that the code had to survive was also low, so the probability

the code would stay with the source was evaluated to be relatively high, at 80 percent.

$$P_{s|m} = 0.80$$

Probability of Release

The probability the bad code is built into the final release is based on the talent of the project manager in spotting vulnerabilities and the backlog existing in the project. Just as during the merge process, a high backlog causes the project manager to limit the amount of time he can commit to reviewing the code before a release. For FlightGear, the project had a high backlog, which increased the probability that the malicious code would stay with the source through release.

The project manager was assessed to be very capable and experienced, which was based on statistics gathered from OpenHub [55]. The website ranks project managers based on their history and performance on prior projects, and the manager for FlightGear had a high ranking of 7 out of 10. Therefore, the probability of release for FlightGear was assessed to be a relatively low 70 percent.

$$P_{r|s} = 0.70$$

Probability User does not Spot Malicious Code

One of the major benefits of OSS is that the user or customer can view the source code before using it, and consequently, the user acts as another layer of defense against malicious attacks. When using a proprietary product, the user is forced to trust the claims of the provider regarding the security and reliability of the software, but for an open source project, a thorough code review by the customer could highlight a bug or vulnerability. The probability of a user finding the malicious code is based on how thorough the user's review is and their experience with the coding language used.

For this scenario, the assumption was made that the user did a cursory review of the source code, but did not spend extensive resources testing before putting the program to use. This assumption led to a high probability that the user did not spot the bad code. With a more

refined review process, the probability could be lowered substantially.

$$P_{u|r} = 0.95$$

Probability of No Flag

A certain amount of time will pass between the release of the code and its first operational use in theater. During this time, the malicious code will still be attached to the source code, which is available for inspection by the general developers on the project and the other users of the program. There is the potential for many eyes to be scanning the code for vulnerabilities during the time between release and usage. The probability that the malicious code survives this stage is based on the length of time between release and operational use as well as the size and capability of the group using and testing the newly released software.

For the scenario, the assumption was made that the code was released after the time period of analysis used in Section 3.5.2, and there was only two months between the release time and the operational use. With the declining activity indicated by analysis of FlightGear's data, especially toward the end of the analysis period, it was assessed that the users and developers would likely not discover the bad code.

$$P_{n|u} = 0.80$$

Probability Malicious Code Initiates

Even if the OSS process fails to identify the adversary's code, the software still has to carry out the functions for which it is designed. There are no perfect software programs, and as such, there is always a chance that they will fail at their given task. This probability is based entirely on the quality of the adversary's code.

For FlightGear, it was assumed that the talent level of the adversary was high and therefore the code was likely to initiate. All initial conditions required by the software package were assumed to occur, so there was a high probability that the code would load and initiate

properly.

$$P_{i|n} = 0.80$$

Probability Code Causes Kill

Finally, there is a probability that even if the malicious code initiates, it will not have the effect on the UAV that was intended by the programmer. This probability is based on the quality of the adversary's code and the design of the UAV. If the aircraft is designed with survivability in mind, the architecture may be able to withstand an attack on its software. This is similar to the traditional probability of kill given a hit on an aircraft. Just because a missile detonates on or near an aircraft does not guarantee that the aircraft will be prevented from carrying out its mission.

In the scenario, it was assumed that no design considerations were taken into account for software survivability on the UAV. It was also assumed that the quality of the malicious code was very high. This led to a probability of 95 percent that the aircraft would be neutralized given initiation of the software code.

$$P_{k|i} = 0.95$$

3.5.4 Overall Probability of Kill

The overall probability of kill is simply calculated using the values for individual event probabilities defined in Section 3.5.3. The equation for the overall probability of kill is then represented by [46]:

$$\begin{aligned} P_k &= P_a \cdot P_{c|a} \cdot P_{m|c} \cdot P_{s|m} \cdot P_{r|s} \cdot P_{u|r} \cdot P_{n|u} \cdot P_{i|n} \cdot P_{k|i} \\ &= (1.0)(0.98)(0.9)(0.8)(0.7)(0.95)(0.8)(0.95)(0.95) \\ &= 0.339 \end{aligned} \tag{3.1}$$

As a result, the probability of survival of the aircraft is calculated by the following equation.

$$\begin{aligned} P_S &= 1 - P_k \\ &= 1 - 0.339 = 0.661 \end{aligned}$$

A 34 percent kill probability is not ideal, and very few military commanders would trust strategic operations to a combat system with such a high chance of attrition. The assumptions made in this scenario are not necessarily realistic, but they do bring attention to the potential of skilled adversaries with access to OSS development processes to jeopardize system software. There are many ways to reduce event probabilities discussed in Section 3.5.3 in order to reduce the probability of kill significantly.

3.5.5 Survivability Enhancements

By performing the survivability analysis on FlightGear, it became evident that many variables influence a project's susceptibility and vulnerability to attack. The health of a program's development community is the primary factor aiding its resistance to attack. The following list includes some ways by which an OSS project could increase its chances to defend against an attack.

Larger Development Community When a large number of developers spend time working on a program, there are more opportunities to discover of a bug or vulnerability, and a large development community also prevents significant backlogs of work from forming. Open source software projects should strive to encourage membership and keep high level programmers working on the software.

Active Community Members A large development community is of no use if the members are not actively participating in code development and review. Open source project managers should utilize project statistics to verify that the members of their development communities are active, and if needed, they should increase activity by setting project goals and ensuring members stay motivated to work on them. One method for increasing activity could be to insert paid staff into open source project communities, thereby generating

interest and support for projects of interest.

Project Maturity When selecting an open source software program for use, project managers should be aware of the maturity of the program. Even if an OSS project has a lot of members, it may not be very secure due to its lack of history and testing. If possible, OSS programs with long histories should be chosen because projects that have existed a long time have likely withstood several iterations, each with improvements and bug fixes.

Low Backlog A good measure for a project's wellness is activity, which can be evaluated based on community members' management of issues. When there are many issues that need to be addressed, developers are required to spend time fixing problems instead of improving the software. Programs with a low backlog should be chosen to ensure proper time is devoted to reviewing code before major releases.

Restricted Development Community One sure way of improving a program's survivability is by restricting entrance to the development community. Websites such as www.forge.mil, which requires a CAC log-in, ensure that the members adding code to the project have had background checks and can be trusted with sensitive material. The downside to restricting access to the community is the creative potential of the software development process is limited.

Larger User Base Users of OSS can be thought of as testers, each with the ability to examine the source code and report issues back to the development community. With more people using the software, there is a better chance that vulnerabilities will be discovered before they can cause harm.

Summary The vulnerability of a UAV to an attack on its software depends on what types of attacks the designers anticipate. In the example of FlightGear, the aircraft designers did not anticipate the software would be attacked. It is always safer to anticipate that the worst scenario will occur. Due to continuing advances in cyber warfare techniques, UAV

designers should always consider the consequences of one or more software modules being compromised. Understanding the competency of the enemy and the threats is paramount.

3.6 Conclusion

In survivability terms, a 66 percent chance of withstanding an attack is not highly desirable, so if FlightGear was being considered for use in a military hardware system, it would likely be turned down for a more resilient alternative. It is important to keep in mind that this one example is not representative of any other OSS program, and therefore general conclusions regarding the viability of OSS for military hardware applications should not be drawn from the analysis in this chapter. The primary motive for Chapter 3 is to prove the concept of survivability as a tool for evaluating open source programs based on their development communities and project statistics. There were many assumptions made in the scenario for FlightGear that do not accurately represent reality, but were inserted for completeness of the analysis. Refinement of the process to assign event probabilities is needed, and further information on this topic is discussed in Chapter 5.

CHAPTER 4:
Data Analysis

4.1 Introduction

An effective survivability analysis requires accurate event probabilities to provide informative results. In the analysis of FlightGear, these probabilities were estimated based on data gathered from the project's host website, www.gitorious.org [53]. This website provided easy access to information such as project issues by allowing visitors to export data listed in tables on the website to a CSV file, however the process to fetch data from the site took a lot of time and effort. The data was also spread out over several tables, which had to be combined onto one sheet after export to facilitate the analysis. Another complicating factor was Gitorious's merge with GitLab in March of 2015, after which the data for FlightGear was no longer available in the form it was originally downloaded.

In this chapter, a new method for data gathering is introduced with the intention to provide a repeatable process which can be used to quickly retrieve data from multiple OSS projects. The scope of this study is primarily focused on the open source software programs used by ARSENL for their swarm UAV research, which are shown in Table 2.2. Most of the programs in the list are hosted by a repository called GitHub. GitHub stores a lot of data on individual projects, including issue open dates, issue close dates, commits per month, lines of code, contributors over time, and programming languages used, but the data cannot be downloaded directly from the website as with Gitorious.

Instead of providing data directly through a graphical user interface (GUI), GitHub uses an application program interface (API) [57] to provide access to data incorporated by open source projects hosted on their website. It was determined that the best way to gather data from the OSS projects used by ARSENL was to develop a software program that could communicate with the API to pull data needed for the analysis. Ideally, an application would have been designed and developed to access data from any OSS project by interfacing with all major hosting repositories. The scope and time constraints of this study limited the development of a program to retrieve information from GitHub, but possibilities for

future application development are discussed in Chapter 5.

4.2 Software Program Development

Before starting from scratch to design a new application, a search was conducted to locate existing software programs with some or all of the functionality required to pull data from GitHub. A program called "Gitstats" retrieved some of the needed data, but it was not fully adequate to satisfy the requirements for this analysis. The program was open source, so an attempt was made to modify the source code so that it performed what was needed, but eventually it was determined that the program would not work properly without significant adjustment.

No existing program was suitable for the job, so a new program was developed to retrieve the data. In order to structure the development process, a short list of requirements for the program was developed, which included the ability to interface with GitHub's website, to access open source project data, to process the data, and to output it to a file. The Python programming language was used to build the code due to its simplicity and its large tool set. The program's source code is provided in the appendix.

4.2.1 Tools

The GitHub API uses the Hyper Text Transfer Protocol (HTTP) to accept requests for information and deliver them to the user, and software projects hosted by GitHub are broken down into a hierarchical format so that all information about the project is hosted under one web page. The data is listed on the web page or linked to the web page by uniform resource locators (URLs). The basic organization of information on the API is shown in Figure 4.1.

Data are organized into three categories, data items, which are singular entities that do not branch to further data, URLs, which lead to web pages representing more information, and data groups, which are lists of information that may include data items, data groups and URLs. To fetch data from GitHub, an HTTP-based program library called `requests` was utilized, which manages the data transfer process and provides tools that help limit the amount of code needed to perform the transfers [58]. Requests allows easy transfer of data from URLs on the API, and it brings the data into the program in the form of objects and

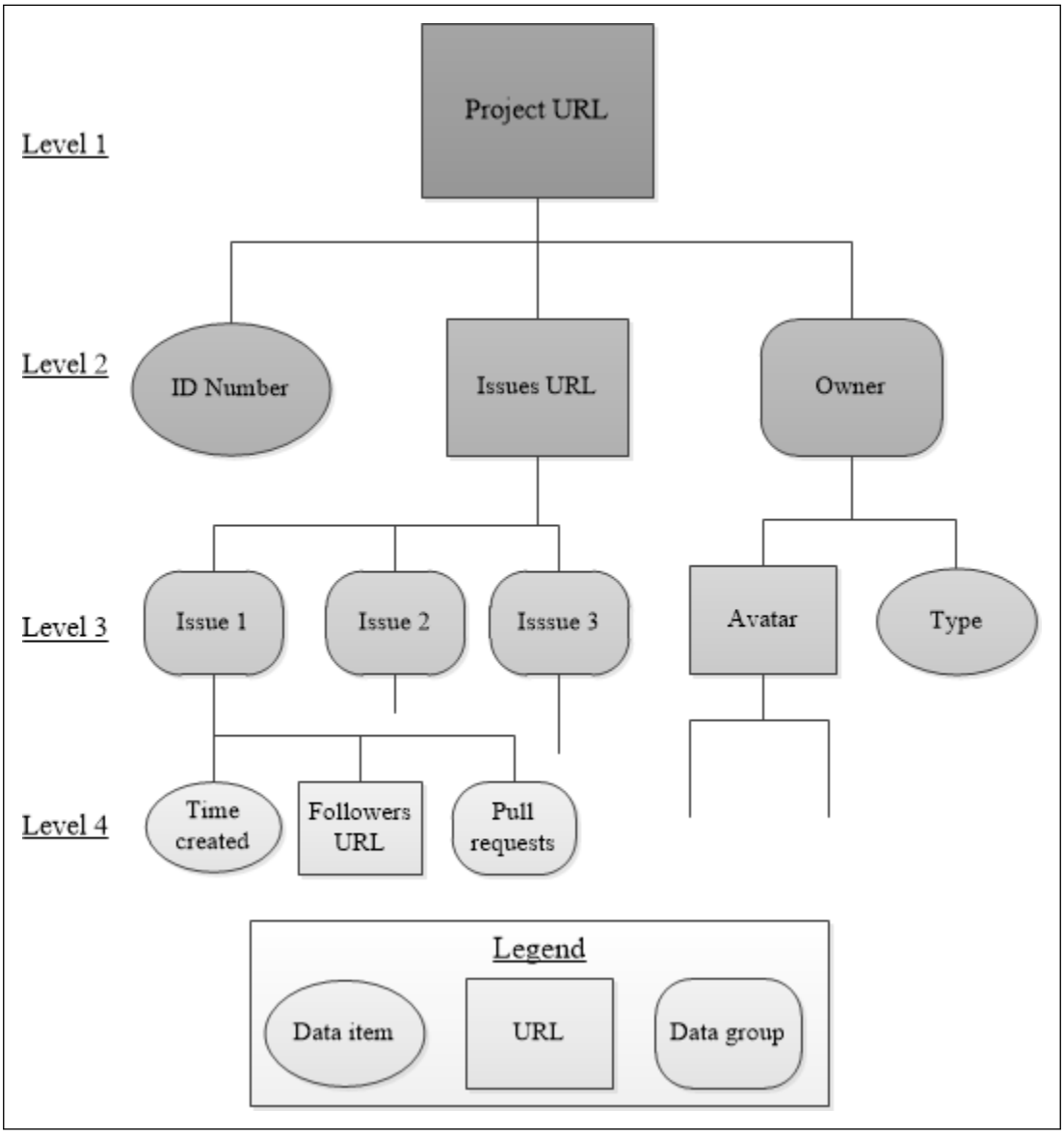

Figure 4.1: General organization of the Github API

lists. The data can then be sorted and plucked from the lists for output.

Once the data was organized properly, another python package called `csv` was used to output the data to a format that could be used for analysis [59]. The `csv` module takes data bundled in lists and reorganizes it into the CSV format, which is understood by spreadsheet programs such as Microsoft Excel.

With the help of these tools, a software program called GitHub Data Analysis was created, and the program allowed GitHub data to be accessed, imported into the program, converted to a different format, and output to a file. From there, the data was examined to gain information about the open source program it was associated with. The types of data that were captured by the program are discussed in 4.2.2.

4.2.2 Data Types

For an open source software project to be successful, it needs to maintain a high level of activity, as outlined in Chapter 2. One method for evaluating the level of activity of a project is by examining its issues, which are similar to change requests and initiated by programmers in the development community. They come in many different forms, including feature requests, bug reports, enhancements, and support requests. If a user finds a bug in the code, he can post it to the project's website as an issue, which is then opened by developers who begin working to find a fix. Once a developer solves the issue, he submits a change to the code and the change is then accepted by a trusted developer, merged into the code base, closed, and archived. This is one example of normal initiation and closure of an issue, but not all issues are addressed immediately due to limited resources or invalid requests.

To evaluate project activity, issue open dates and issue close dates were obtained from the API. These dates provided information about how often issues were initiated and how quickly they were resolved, and they indicated the total number of issues that were open at a given time. Issue data allowed estimates to be made about activity levels and overall project health, just like it did for the case study about FlightGear, discussed in Chapter 3.

While the GitHub Data Analysis program worked well, it is not complete, and future iterations need to analyze more data from open source projects. There is a lot more information available on repository websites that reflects the quality of OSS projects, such as commits over time, number of developers, lines of code, and programming languages used. Due to time constraints, the program only captured data on project issues, but future development considerations are discussed in more detail in Chapter 5.

4.3 Example Analyses

4.3.1 ArduPilot

The first project to be analyzed using data collected by the program was ArduPilot, an autopilot for UAVs. It is the most popular open source autopilot on the market [60], and it incorporates approximately 180,000 lines of code provided by over 200 contributors. Generally, the project is thought to be healthy, with high activity and a strong development community [60].

Github Data Analysis pulled data for 2323 issues from the project repository. Issue open and issue close dates per month were then graphed over the past two years, shown as a scatter plot in Figure 4.2.

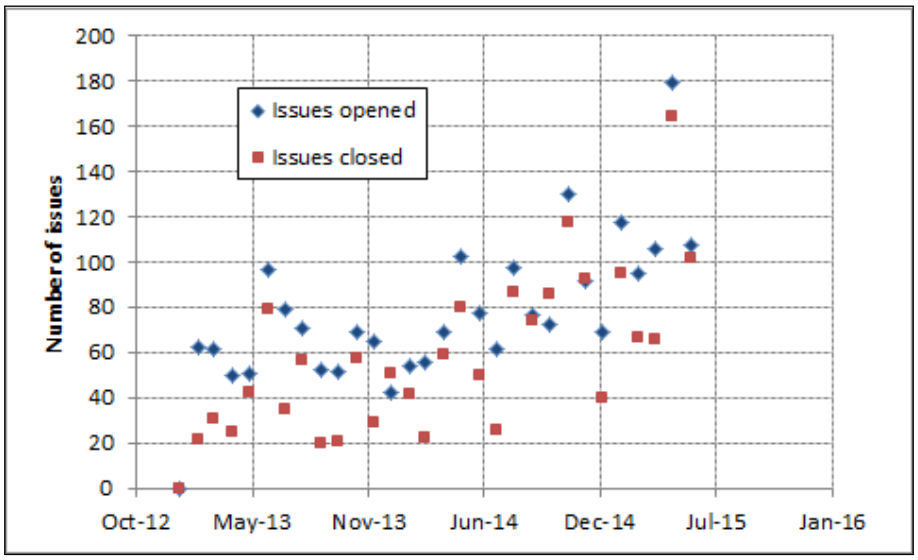

Figure 4.2: Ardupilot issue open and close dates over time

Each data point in Figure 4.2 shows the number of issues opened or closed in that month. The first indication that the data provides is that there was an increasing trend for issues being opened and issues being closed as time progressed during the analysis period. This is one indicator of good project activity, as higher issue counts equate to more software developers working on the project. The indication of higher activity levels in Figure 4.2 represents increased health for the project.

It is helpful to visualize the general movement of the number of issues being opened and closed over time, which can be seen better by adding trend lines to the plot, as shown in Figure 4.3.

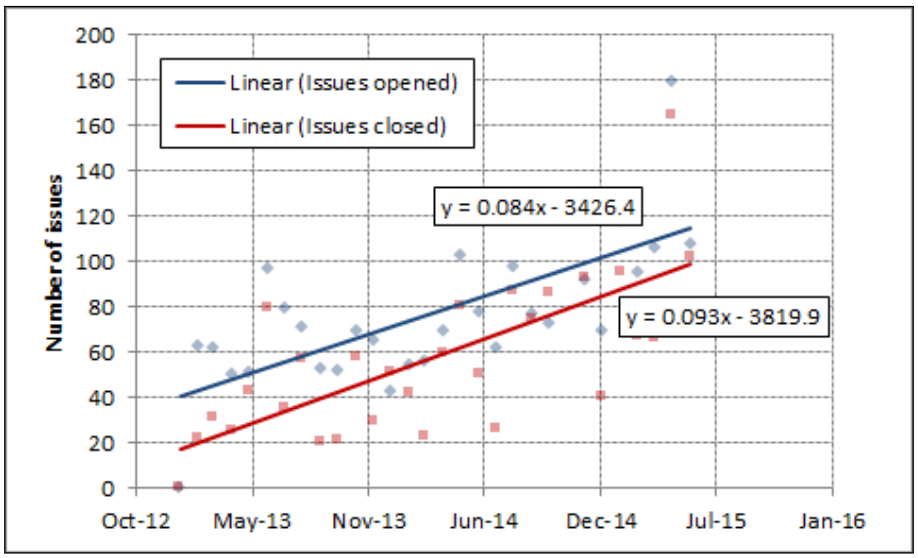

Figure 4.3: ArduPilot issue open and close dates with trend lines

The slopes of the data shown in Figure 4.3 are close, but the line for issues being closed is slightly steeper than the line representing issues opened (0.093 versus 0.084), which is a good sign for the project health of ArduPilot. A positive trend in opened issues means that users and developers were uncovering problems in the program at an increasing rate, and a positive trend for closed issues indicates solutions were developed for the problems at an increasing rate as well. Issues were also being closed at a faster rate than they are being opened. Going forward, this could mean that the project would see a decreased number of total issues that remain open, or a decreased backlog of issues.

The backlog of issues is another indicator of project health, as discussed in Chapter 3. The backlog for ArduPilot is shown in Figure 4.4.

Over two years, the number of issues remaining opened increased from zero to around 600, and this could have been due to a number of reasons. Some issues may not have been addressed due to a lack of developers who had time to work on them. Others may have been technically difficult problems for which a solution had not been discovered, or the issues may not have represented a problem, fix, or bug that needed to be corrected to

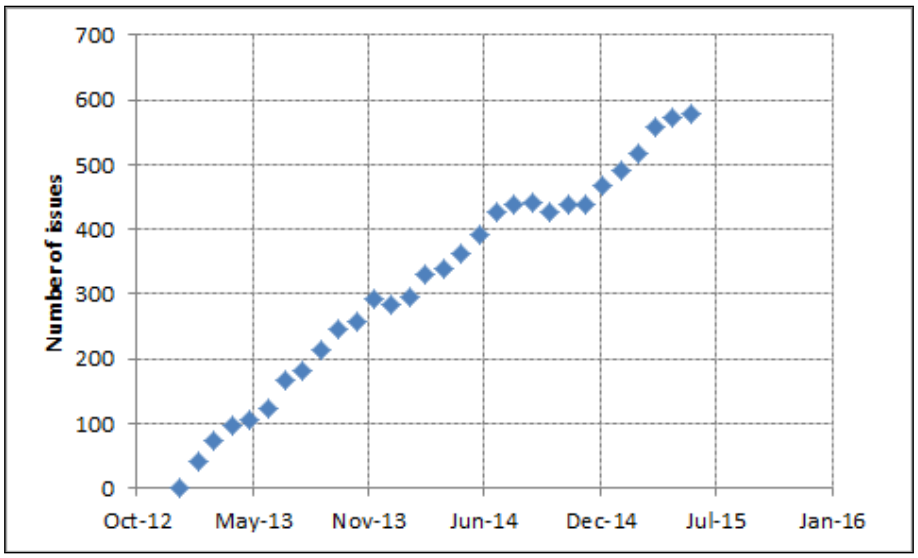

Figure 4.4: Total ArduPilot issues that remained open over time

meet the project's requirements. Issues can be posted by any members of the development community, and they are not necessarily vetted by caretakers to confirm their worthiness to be tracked. Furthermore, "Issues" also includes feature requests, and if bug fixes are being given priority then feature requests can fall by the way-side. Also, if no one is actively monitoring feature requests, there could be a fair number of duplicates as different users request the same features.

It should be noted that only data relating to issues was available for the analysis of ArduPilot, as well as the other OSS programs in this section, due to stalled development of the data gathering program required for the analysis. Conclusions being made in Chapter 4 are tempered by the understanding of incomplete data, and full analysis of these programs is remanded for future work, as discussed in Chapter 5.

4.3.2 MAVLink

Another open source program used by ARSENL is MAVLink, which provides communication protocol for micro air vehicles [61]. MAVLink was analyzed in the same way as ArduPilot. Figures 4.5, 4.6, and 4.7 show the the raw issue data, trend lines, and backlog for MAVLink.

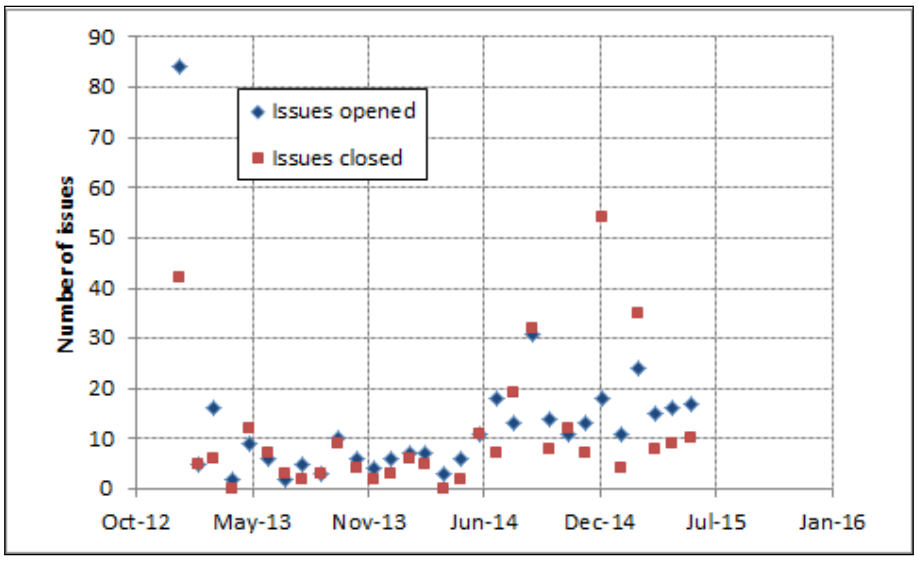

Figure 4.5: MAVLink issue open and close dates over time

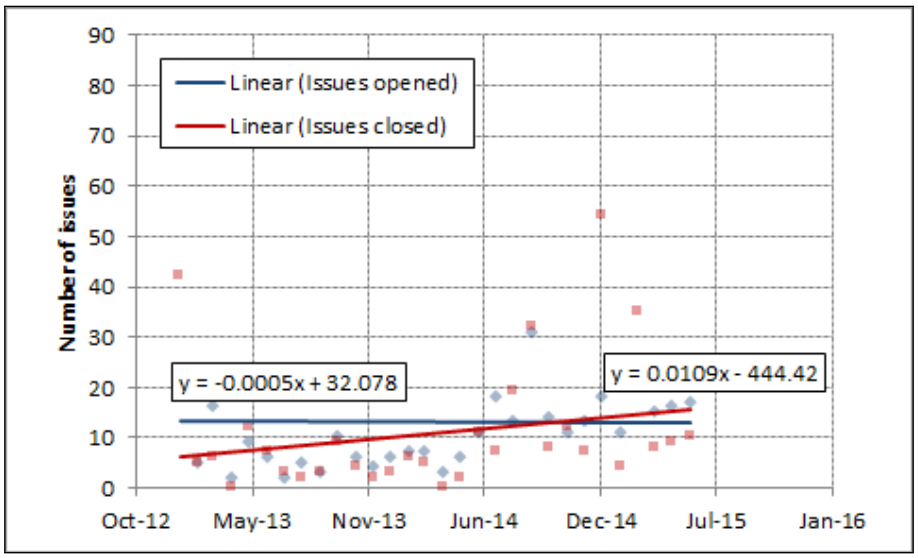

Figure 4.6: MAVLink issues open and close dates with trend lines

The trend lines indicate an increase in issues being closed over time, which is similar to ArduPilot. The data showing issues opened over time has a nearly flat slope that is slightly negative, which is mainly due to the first data point in the series being abnormally high. The abnormally high data point in MAVLink's data set for opened issues could be the result of a large number of issues introduced at the start of the project, when the code was first established. If the first data points are removed, the trends for MAVLink look

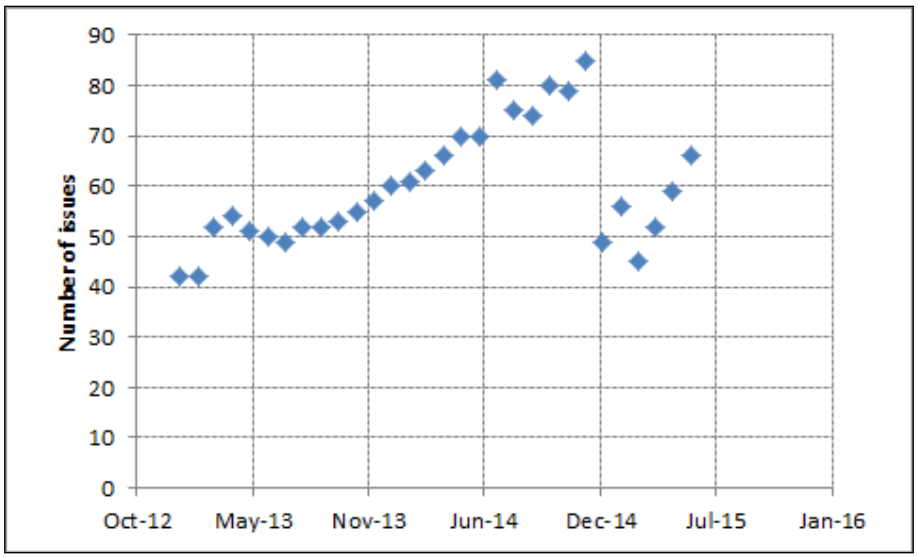

Figure 4.7: Total MAVLink issues that remained open over time

similar to ArduPilot. The issue backlog for MAVLink reflects good project health during this time frame because the number of open issues rises initially but falls toward the end of the period, indicating that most of the issues were addressed and fixed. Like ArduPilot, MAVLink is identified as a project with high activity on OpenHub [62]. Analysis of the data gathered by the GitHub Data Analysis program indicate a similar conclusion, that MAVLink has good activity and would likely withstand a malicious software insertion better than a program like FlightGear.

4.4 Collective Analysis

Several other open source programs were analyzed for comparison. Table 4.1 shows the consolidated results, and Figure 4.8 shows trend lines for each program's issue open dates and issue close dates.

Figure 4.8 allows the health of each program to be compared by examining the difference between slopes of the trend lines representing the number of issues opened over time and the number of issues closed over time. Ideally, the slope for the number of issues opened should be lower than the slope for the number of issues closed. This would indicate the project is able to limit the percentage of issues that remain open over time. For the projects analyzed, all but two (MAVProxy and PX4 Firmware) have this characteristic.

Program	Opened dates trend line slope	Closed dates trend line slope	Backlog slope	Backlog at finish
ArduPilot	0.0840	0.0930	0.6170	579
MAVLink	0.0165	0.0179	0.0225	66
MissionPlanner	0.0132	0.0147	0.3397	242
MAVProxy	0.0123	0.0085	0.0247	35
PX4 Firmware	0.1220	0.1137	0.1403	161

Table 4.1: Issues analysis results for several open source projects related to UAV software development

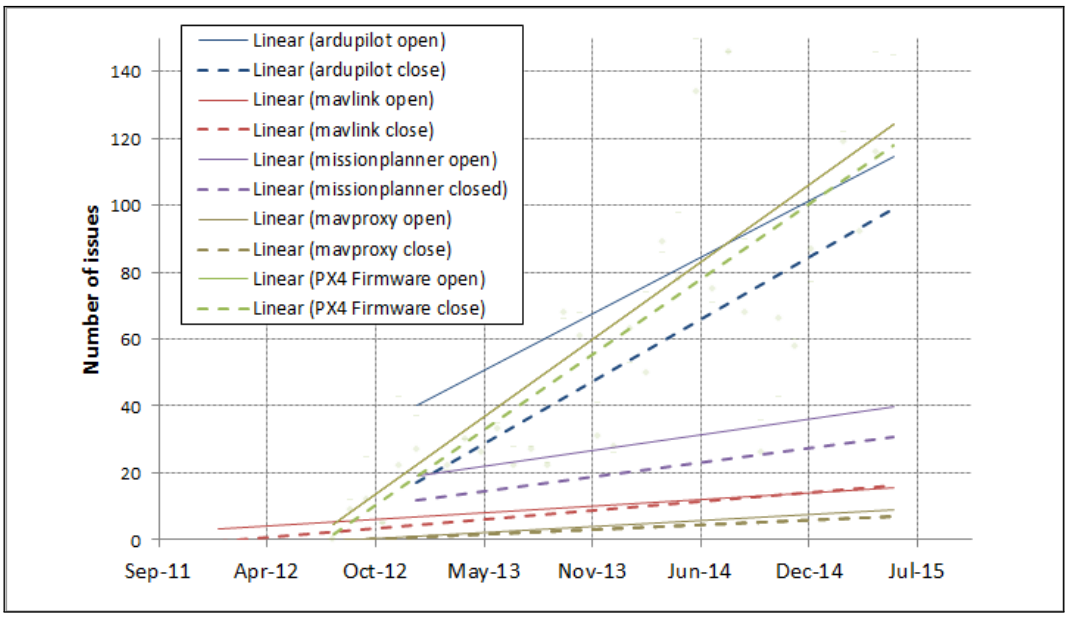

Figure 4.8: Number of issues opened and closed over time for six OSS programs

Comparing the slopes of the trend lines between programs also provides information about the relative activity level of each program. Open source programs with high slopes indicate there were progressively more work activity events at the end of the analysis cycle versus the beginning. The large end number could be attributed to an increase in activity events being performed by each member of the development community, but more likely it is due to an increase in the total number of developers working on the project. For example, the slope of the trend lines for PX4 Firmware are large (about 0.12), which means the project activity for that particular program increased rapidly over the time period analyzed. In contrast, the slopes for MAVLink are small (0.016), which indicates MAVLink experienced

similar activity levels at the beginning and end of the analysis period. A small slope is not necessarily a bad sign, but increasing activity is a reasonable indicator of future project health.

Finally, one of the most important observations is the distance between lines for individual programs. This indicates the difference between the total number of issues opened and closed over time, which contributes to the project backlog discussed in Section 4.3.1. Figure 4.9 shows backlogs for the same six programs shown in Figure 4.8.

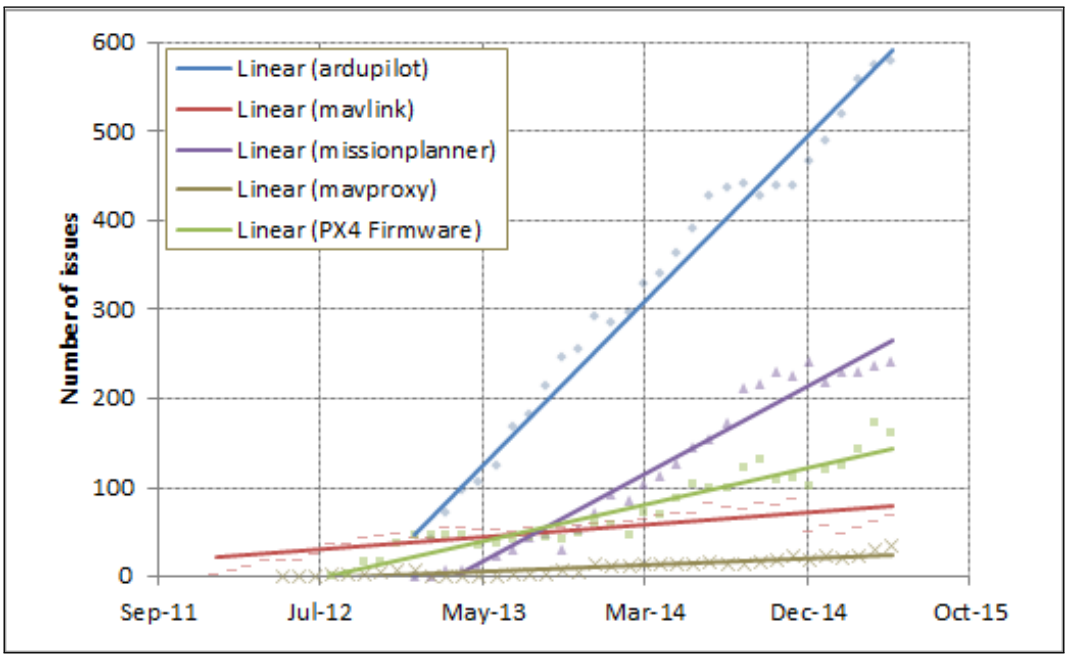

Figure 4.9: Issue backlogs for six OSS programs

Ideally, open source projects would maintain constant backlogs, closing issues at the same rate they are opened. For the projects analyzed, a large separation between lines in Figure 4.8 led to a higher backlog. This was due to the higher number of issues opened as the projects progressed, as compared to the number of issues closed. Figure 4.8 shows a large separation between trend lines for Ardupilot which is the reason for its steep slope in Figure 4.9. In contrast, despite the steep slopes of the trend lines for PX4 Firmware, shown in Figure 4.8, the lines are close together. The backlog for PX4 is therefore sloped mildly as compared to some of the other projects like ArduPilot. Theoretically, lower backlogs are good for open source projects as they indicate a low workload for developers as programs

prepare to release their products to users.

4.5 Conclusion

The issue data gathered for this analysis was similar to that obtained for the FlightGear case study, and the analysis of issues provided results with similar characteristics. Data for five projects was analyzed, and the results indicated that the projects had good activity levels, as each one experienced an increase in issues opened over time and issues closed over time during the period analyzed. While the analysis of issues provides some indication of a project's health and its ability to withstand an attack by an adversary, the examination needs to go into more depth (including interactions between different metrics) to make plausible conclusions about a program's survivability. GitHub provides more data such as commits and developer information, and Chapter 5 explores methods for increasing the benefit of this analysis by gathering a larger representation of data for projects being evaluated. It is presumed that future analysis techniques will allow for better predictions of the event probabilities used to calculate a program's survivability.

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusion

While proprietary software companies compete to gain market share, open source software continues to survive and flourish. Its growing popularity draws support and funding from large commercial companies who see the benefits of community-developed, open software products. OSS also helps the DOD carry out its mission in many different capacities, including information security, database management, and software development.

While technology development in the military often lags industry, the Department of Defense understands the value of OSS. The DOD has indicated that open products should be given consideration alongside proprietary options when choosing between software alternatives. Open source software is important now, and signs indicate that its prominence will increase in the future.

An important consideration for the DOD is determining how it will secure its hardware and software systems against cyberattacks. The expansion of cyber warfare is evident by the military's increased posture against such threats, and it is important to evaluate the ability of military software to defend against cyber aggression. In small UAVs, software is both vulnerable and susceptible to attack, and large hardware systems including aircraft and ships are no different. Software needs to be developed with this threat in mind.

The concepts developed in this thesis address the question of software survivability. Methods for analyzing the survivability of software were developed and discussed on several OSS projects used on small UAVs, and the analysis showcased the potential of survivability concepts to identify ways to reduce the probability UAVs are compromised by an attack on their software. To introduce the survivability analysis process, a program called FlightGear was analyzed by gathering data for the project, including issue dates, commit dates, and developers over time. The OSS development process was segmented into several events, and each of the events was assigned a probability based on heuristics and informa-

tion gleaned by analyzing data for FlightGear. After probabilities were assigned for each event, the overall probability of survival for a small UAV utilizing the FlightGear software was calculated. Based loosely on the data, along with several assumptions for the scenario, it was determined that FlightGear only had a 60 percent chance of surviving and malicious attack.

The results of the FlightGear analysis tested the concept of using traditional survivability to evaluate the quality of the development process for an open source project. The data used for the analysis was plucked from the websites manually, which took a substantial amount of time. It was determined that in order for the analysis to be worthwhile, data needed to be collected much more efficiently due to the large number of OSS projects in need of analysis. A single military hardware system can contain many software programs, and to obtain data for those programs manually by visiting websites is inefficient and cumbersome. To solve this problem, a software program was designed to help facilitate data gathering for open source projects.

Most of the open source programs used by ARSENL for UAV development are located on the GitHub version control repository, so the program was designed to operate specifically with this repository. The GitHub API was used as an interface for the GitHub Data Analysis program to reach and obtain the data. Once the program had data, it organized it and output it to a spreadsheet, where it could be analyzed using the Microsoft Excel spreadsheet program.

Ardupilot was one of the open source software projects analyzed, and during the time period of the analysis, ArduPilot showed high activity. The activity level was based on the number of issues being opened and closed over time, and both categories of issues increased during that span. This meant that the project experienced increasing levels of developer involvement as time progressed. The average number of issues being closed out on a monthly basis was lower than the number of issues being opened. This meant that the project was also experiencing a significant buildup of issues waiting to be closed. Depending on the nature of the issues remaining open, this could mean that the project was over-stressed at some points of the analysis cycle, but general trends in the data still suggested that Ardupilot maintained high activity levels, which indicates good project health.

By this analysis, Ardupilot is assessed to be a healthy OSS program, and this assessment is supported in an independent analysis performed by Black Duck Software Inc. [60]. Healthy OSS projects are more resistant to malicious attacks than projects with small user communities and minor activity levels, therefore, event probabilities described in Section 3.5.3 would be lower than average for a project like Ardupilot. This means that an aircraft using Ardupilot would have a higher chance of surviving a cyberattack on its software.

The extensive research conducted for this study, combined with the results obtained by the survivability analysis, provide some general guidance on ways which programs can be guarded against attacks on their development processes. The research on these topics is not exhaustive, however, and there is availability for expanded analysis and exploration into these areas and related areas that could improve and validate the survivability analysis procedures presented here.

5.2 Future Work

Nominally, the presented analysis would contribute more information to the assignment of probabilities for individual events in the scenario by examining more programs, more data, and more features of the projects. As discussed in Chapter 3, characteristics such as the issue backlog disclose information about an open source project's ability to withstand an attack. The demonstrated methodology for evaluating a program's survivability helps validate event probabilities, but the analysis could be expanded and refined to provide more evidence for quantifying the probabilities. The need for more research and future work is discussed in Section 5.2.

The work performed in this thesis led to new ideas about software development and program analysis. There are many more topics that should be considered and explored to understand how development methods (especially open source methods) can be improved. Potential avenues for future work are discussed in the following sections.

5.2.1 Data Requirements

The GitHub Data Analysis program only retrieved projects' issues from their repository, but GitHub provides a tremendous amount of additional data on each open source project.

Characteristics such as the number of lines of code incorporated in a project, the number of commits made per month, the commit frequency, the number of developers in the community, and the languages used in the program can all be found on GitHub. Different repositories, such as SourceForge, GitLab, BitBucket, and Codeplex, may offer even more data pertaining to the OSS programs hosted on their websites. The quality of the survivability techniques discussed in Chapter 3 will increase with a better understanding of the data associated with OSS projects being analyzed. The following sections list the benefits of acquiring specific data types.

Development Community

The development community working on an open source project could provide ample information about project health. Programmers are required to have a profile (at minimum a username and email) in order to work on projects, and they often provide more information such as their name, their background, their work history, and their interests. This information could provide insight to the health and well-being of a program. Of special importance are the trusted developers, or caretakers of open source software, as they are the few programmers who have special permissions to approve new software to be merged into a repository's code base and release new versions of the software to the users. Their backgrounds are especially relevant because they are so involved in producing the programs. Data points such as the number of other open source projects they have managed (a measure of experience) could predict the future success of their current open source projects. Other attributes of the development community that could be explored are the number of general developers working on the project, the profiles and histories of the general developers, and amount of time developers spend working on projects.

Lines of Code

The number of number of SLOC incorporated in an open source software project is another category of data to explore. Projects with a large code bases are generally more mature, if only because it takes time, people, and resources to write code. A project with many lines of code must have existed long enough to gain popularity, otherwise it would have failed due to lack of support. Programs need to be useful for developers to be motivated to work on them, and if they do not have much value, they will not likely remain active. An

existing source for data on OSS lines of code is Black Duck's OpenHub website [55]. The website provides graphical information representing the total lines of code in projects and the rate the code was developed. Use of this metric, however, comes with understanding that since SLOC is not comparable between projects written in different languages, it may provide less utility than other metrics which are applicable across all types of projects. It may be useful to determine a relationship between SLOC and program language before adding SLOC to survivability analysis procedures.

Commits

Commits are an indicator of project activity. Viewing the number of commits made over time can show developer activity level, and there are many ways which commit data can be manipulated to contribute to various metrics. Total commits for a project can be examined to get an idea of overall project activity, and individual commit levels provide information about the primary developers, such as their work consistency, dedication to the project, and coding capability. One commit-related metric analyzed by Kolassa *et al.* is the time between code contributions [63] for individual developers. Their research determined that projects with lower commit frequencies were not as successful as projects with higher commit frequencies. This theory could be expanded in future work to determine how commit frequencies affect OSS survivability. Commits are a fundamental part of the open source development process. They could be important for defining metrics for survivability.

Programming Language

Each software program is written in a certain programming language, some examples being C++, C, Python, JavaScript, Java, XML, and Matlab. The language open source projects use could reflect other characteristics of the code such as the security, the functionality, and the performance capability of the projects. A separate analysis would be required to determine if and how programming languages affect software programs. The languages used could also provide information about the demographics of the developers writing the program. If there are language preferences among known groups of developers, this information could be used to categorize the community of people working on certain projects and determine their motivation and capability for for working on them.

Issues

Issues were captured in this analysis, but there is more information that can be gleaned from the data than what was gathered. There are many types of issues that can be generated for a project, and bugs and vulnerabilities are just one type. Other project concerns such as feature requests and integration questions show up in the repositories as issues as well. Some issues provide significant information about project health or activity, while others providing less fruitful statistics. One way that GitHub allows project managers to filter their project's issues is through labels, which allow categorization of each issue tracked by in a project. If a project manager applies labels, users and developers can gain general information about issues by simply looking at a glance. Labeling also allows programmers and managers to prioritize which issues should be handled first, and issue labels could be used to filter the unimportant issues out of the picture before performing a survivability analysis.

Issue data could also be explored to determine correlations between developers, commits, and issues. It may be useful to examine who makes a lot of commits, who generates issues, and who closes issues, in order to discover which programmers achieve an equal balance between writing code and fixing problems. This balance may be a desirable characteristic for a project developer, or it may have little effect at all. Monitoring the breakdown of commit work versus issue work could help determine how it relates to the success of a project.

Open Source Repositories

There are several popular open source repositories used by OSS project managers. Only GitHub was analyzed during this research, but others include GitLab, CloudForge, Bitbucket, Launchpad, and SourceForge [64] [65] [66]. Other hosting platforms need to be reviewed and included in the data gathering process so that survivability techniques can be applied to all programs of interest.

Not all repositories use the same framework for revision control. Git is a popular revision control system used by GitHub and GitLab, among others, but other control systems include SVN, Bazaar, Mercurial, and Veracity [67]. Each has their own way of maintaining repositories, which affects how their databases are accessed. The Github Data Analysis

program needs to be expanded to handle other revision control systems, and this will allow a broader sample of open source projects to be analyzed.

5.2.2 Common Vulnerabilities and Exposures

The Computer Vulnerabilities and Exposures (CVE) list is published by the MITRE Corporation, and it is "...a dictionary of common names for publicly known information security vulnerabilities" [68]. When software vulnerabilities in computer programs are discovered, they are added to the list produced by MITRE, and computer security companies such as Symantec use the list when designing software to defend against attacks.

The CVE list could be used as another tool to identify whether certain open source programs are appropriate for military applications. Data from the list could be gathered and added to the data from open source repositories. If vulnerabilities were identified for an open source project, they could be compared to vulnerabilities in proprietary projects to determine how they might affect the operation of the program, and this could help define event probabilities for a project's survivability analysis.

The presence of known security flaws (which have been fixed) does not necessarily mean that a program is not secure, but it likely means that more people are looking at the code. All code has bugs, and finding the vulnerabilities depends on how hard users look for them. A project with flaws that have been identified has likely been reviewed and tested extensively, and this is good for any software project, including OSS, because scrutiny leads to better product development. The CVE list is essential to a good security evaluation system, and it would help substantiate an OSS software selection tool based on survivability.

5.2.3 Survivability

The survivability analysis introduced in Chapter 3 focused on one OSS program. It presented a means for calculating the survivability of the software program to withstand an attack against a fictitious enemy. It is a good baseline for future survivability analyses to expand on, but several portions of the study could be improved to evaluate other software products. Potential topics to help improve the survivability analysis are discussed in the following sections.

Threat Analysis

For FlightGear, a threat scenario was designed as an example, which was completely fictitious, and no research was conducted to assess actual threats to OSS used in UAVs. One of the fundamental principles of survivability is the threat to the object being analyzed, and in order to defend against a threat, its capabilities must first be understood [69].

Primary threats designed to fight military aircraft are surface to air missiles, anti-aircraft artillery, and air-to-air weapons. An aircraft's survivability depends on what threat it is going against, who is operating the threat, and the environmental factors surrounding the scenario. This is why extensive live-fire testing is performed for all major military systems [70]. Software could benefit from the same type of testing if general guidelines and requirements for procedural guidance were established. Conventional threats to aircraft have been studied and documented for decades, but cyber warfare is a much younger movement, so threats have not been evaluated or cataloged in as much depth. Each potential enemy should be evaluated for their ability to conduct attacks on software, including their ability to infiltrate open source software programs as in the scenario presented in Chapter 3. Once known threats are established, individual programs could be evaluated on their ability to defend against them.

Probability Assignment

Another cornerstone of survivability analysis is the correct identification of probabilities associated with events in a scenario. E3A analyses help identify specific events that would occur if an attack scenario were to take place. To determine the overall probability of survival against the threat, probabilities for each event in the E3A are computed and inserted into the equation for the probability of kill, as in Section 3.5.4. The overall probability is only valid if the individual event probabilities are sound, so the event probabilities are what justify the validity of the survivability of a program.

In order to gather accurate event probabilities, the data needs to be updated and accurate, and the methodology needs to be good. This thesis hypothesizes that repository data relating to an open source project reflects the project's quality, but this hypothesis needs to be verified. One option is to research open source evaluation techniques, and several models have already been designed to help determine the attributes that successful projects

share [71] [72] [73] [74]. These models could be used to verify the methodology of future revisions of the GitHub Data Analysis program, and could give credibility to the assignment of event probabilities.

OSS Project Differences

There is a wide range of open source project sizes, and some large project have been under development for many years. These projects have big development communities and extensive data that can be analyzed, but other programs are smaller, with little background or history to draw from. There were five programs analyzed in Chapter 4. ARSENL uses many more programs than the ones evaluated, but some of them were too small to be analyzed by the GitHub Data Analysis program due to their issue counts, which would not have provided useful information.

OSS programs with short histories should not be automatically disqualified for use, as there are other indicators of project health other than issues. It may not be feasible to compare all open source projects in the way demonstrated in Chapter 4 due to their widely differing characteristics. Further research is needed to recommend other methods for categorizing and analyzing smaller OSS programs. Once the survivability of individual programs being used by a system is identified, an integrated analysis of all programs would be possible.

There are also differences between programs due to their functionality. Section 2.4.2 discusses the types of software used on UAVs. Some programs have a much bigger impact on UAV survivability than others, for example, ordinance control software, if compromised, could have a dramatic affect on an aircraft's operation. A program used to capture and record routine flight data would not have as big of an affect, but if the two programs were integrated in such a way that an adversary could access the ordinance control software through the data recording software, he could potentially cause serious damage by simply adjusting the flight recorder. To fully understand the survivability of open source software used in military systems, dependencies between software programs should be studied.

Proprietary software

The concepts of survivability for OSS could be adapted and applied to proprietary software. The biggest hurdle to this task is acquiring data from commercial companies about their

programs. It is not in the nature of closed software developers to release information about the products they create. In contrast to OSS, information about the development process of proprietary software is rarely available, and neither are statistics such as issues, commits, and number of developers working on a project.

Survivability of proprietary software could help determine a recommendation for the type of project to use for a specific application. It would also be important if a system utilized both proprietary and open source software, which is common. By identifying the vulnerabilities of all programs being used, a complete picture of a system's probability of survival could be developed.

5.2.4 Advanced Analysis Model

The software program that was developed for this thesis streamlined the process for gathering data. It allowed multiple entries for issues to be collected simultaneously. The program wrote the data to a CSV file format. The data needed to be analyzed manually using a spreadsheet program. It would be ideal to design a program which grabbed more data and processed it immediately.

A domain analysis should be conducted to understand more about how similar programs work. This could be followed by a standard stakeholder analysis and functional breakdown for the potential computer program. Requirements should be developed to provide guidance through the design process and maintain consistency during development.

The following sections describe recommendations for a follow-on program.

Data Fetch

The first need for a new program is a wider array of data. Section 5.2.1 describes several types of data that could be useful for OSS program evaluation. The data needs to be taken from GitHub and other websites that host OSS repositories. Developers need to research the different revision control systems used by the host sites. After obtaining the data from the repositories, it would be stored and managed by the program so it could be used for output.

Data Manipulation

After storing the data locally, it needs to be processed. This could include combining and separating data, performing mathematical analysis, applying statistical techniques, and omitting unusable data. The processing would be where the most effort would be required. After the data is processed, it needs to display output to the user.

Program output

The output of the program should be tailored to the needs of the stakeholders. One of most useful products of the code utilized in the thesis was the graphical representation of the issues over time. This was a great tool for visualizing attributes of individual projects and comparing multiple projects at once. Graphics of some sort will likely be required for follow on projects. They present the data concisely and allow decision-makers to better understand information buried in the numbers.

The output could also provide direct recommendations for deciding between open source alternatives. Decision-making analysis could be employed to help with these recommendations. The program may need to accept input from the user to define the scope of their needs before performing the analysis and presenting the output. The program should backup any recommendations by justifying them with data.

5.3 Summary

There are many directions in which this analysis could progress. Open Source Software projects provide data which is generally open and available to study. This is one of the great benefits of OSS. Recently, repository websites have improved, with many providing their own graphs which show statistics about project health, activity levels, and popularity. There are also organizations dedicated to the study and evaluation of open source projects, for example, Black Duck Inc. performs analysis on many of the open source projects in today's market. As the popularity of OSS continues to increase, so will the need for critical analysis of its capability. This thesis provides a starting point for survivability to be used as a tool in the analysis and evaluation of OSS.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Program Code

The GitHub Data Analysis program source code is provided to support potential future work. The program is actually broken into two text files, githubIssuesAnalysis.py and analyzeGithubData.py, which are both shown here. The programs can be downloaded from the following website: `https://yoda:18080/dcumming/github_analysis` or requested from the author or his adviser. Chapter 5 can be referenced for ways to expand and improve the program.

A.1 githubIssuesAalysis.py

```
"""
Script to get issue information from GitHub using GitHub API

Uses basic 'requests' package for Python to handle HTTP(S) requests (per GitHub API)
    REF: http://stackoverflow.com/questions/10625190/most-suitable-python-library-for-github-api-v3

Author: T.H. Chung
"""

import numpy as np
import requests      # http://docs.python-requests.org/en/latest/user/quickstart/
                     # http://docs.python-requests.org/en/latest/
                     # http://www.pythonforbeginners.com/requests/using-requests-in-python
import json
import ssl
import os
import datetime

# Define the GitHub repository to be inspected
name_input = raw_input("Please enter full name of GitHub project (e.g. 'diydrones/ardupilot'):")
repo_name = 'https://api.github.com/repos/' + name_input
projectNameList = repo_name.split("/")
projectName = projectNameList[-1]
today_str = datetime.date.today().strftime('%Y%m%d')
```

```
25  dataFile = projectName + '_data_' + today_str + '.txt'
26
27
28  # 1. Inspect appropropriate path to records text file
29  # 2. Check if file with name is in that directory
30  recordFileExists = os.path.isfile(dataFile)
31
32  # 3. If yes, "load" text file into script variable
33  if (recordFileExists):
34      # 3. "load" text file into script variable
35
36      print "Data file exists. Use 'analyzeGithubStats.py' to analyze %s" % dataFile
37
38      # Continue with processing
39      # Example: Access the first record with 'd[0]'
40      # Example: Access the 'created_at' time of the first record with 'd[0]['created_at']'
41
42  else:
43      # 4. If no, execute the "requests" code to get the actual data from the URL
44      r = requests.get(repo_name)
45
46      if(r.ok):
47          # Get the number of open issues for this repository
48          num_open_issues = r.json()['open_issues_count']
49          print("Number of open issues: %d" % num_open_issues)
50          print ""
51
52      # Get header information regarding issues for the specified GitHub repository
53      # - Fetch up to 100 issues per page (maximum allowable, to minimize number of request calls
            )
54      # EX: test = requests.get('https://api.github.com/repos/diydrones/ardupilot/issues?page=2&
            per_page=100')
55      test = requests.get(repo_name + '/issues?per_page=100&state=all')
56      if(test.ok):
57
58          # Determine number of pages, using header information
59          # REF: https://developer.github.com/v3/issues/#list-issues-for-a-repository
60          # REF: https://developer.github.com/guides/traversing-with-pagination/
61          # Contained in the header
```

```
62          link_to_last_page = test.links['last']['url']
63
64          # Parse 'link_to_last_page' string to find total number of pages [check]
65          # EX: 'https://api.github.com/repositories/7512484/issues?per_page=100&page=6'
66          parsed_url = link_to_last_page.split("=")
67          num_pages = int(parsed_url[-1])
68          print link_to_last_page
69          print num_pages
70
71          repoItem = []
72          # Initialize variable to store all issue times
73          issue_times = []
74          create_times = []
75
76          # Iterate over all pages and get listing of issues
77          for page_num in range(1,num_pages+1):
78
79          # for page_num in range(1,2+1):
80              issue_page = requests.get('%s/issues?page=%d&per_page=100&state=all' % (repo_name,
                    page_num))
81              print issue_page
82
83              # Capture the issues in JSON format to make it easier to parse
84              # Same as 'issue_page.json()'
85              repoItem.append( json.loads(issue_page.text or issue_page.content) )
86              print ""
87
88              # Iterate over all issues on this page and print out the desired field
89
90          # Flatten the list (so it's only one-dimensional)
91          # REF: https://coderwall.com/p/rcmaea/flatten-a-list-of-lists-in-one-line-in-python
92          flatrepo = [item for sublist in repoItem for item in sublist]
93
94          # DUMP TO the text file so I don't have to fetch in the future
95              # REF: https://docs.python.org/2/library/json.html
96              # REF: http://stackoverflow.com/questions/12309269/write-json-data-to-file-in-
                    python
97          with open(dataFile,'a') as outfile:
98              json.dump(flatrepo, outfile, indent=4, separators=(',', ': '), sort_keys=True)
```

```
99
100 """
101 Some nice references :
102
103  Interesting  graphics  or  comments on metrics
104 [ ] http :// webapps.stackexchange.com/questions/41336/github-new-fixed-issues-statistics
105 [ ] http :// blog.comindware.com/solutions/bug-tracking-software/
106 [ ] http :// sqa.stackexchange.com/questions/1888/what-bug-statistics-are-most-useful-why
107 [ ] https :// github.com/StephenOTT/GitHub-Analytics
108 [ ] https :// www.dropbox.com/s/xrr37sefj0xpr93/Ruby%20Tuesday%20-%20Github%20Analytics.pdf
109
110  Interesting  software
111 [ ] https :// github.com/michaelliao/githubpy
112 [ ] http :// stackoverflow.com/questions/10625190/most-suitable-python-library-for-github-api-
        v3
113 [ ]
114 """
```

A.2 analyzeGithubData.py

```
"""
Script to output issue information obtained by githubIssuesAnalysis to CSV

Uses basic 'requests' package for Python to handle HTTP(S) requests (per GitHub API)
    REF: http://stackoverflow.com/questions/10625190/most-suitable-python-library-for-github-api-v3

Author: T.H. Chung
"""

import numpy as np
import requests      # http://docs.python-requests.org/en/latest/user/quickstart/
                     # http://docs.python-requests.org/en/latest/
                     # http://www.pythonforbeginners.com/requests/using-requests-in-python
import json
import ssl
import os
from pprint import pprint
import datetime
import csv

# Define the GitHub repository to be inspected
name_input = raw_input("Please enter full name of GitHub project (e.g. 'diydrones/ardupilot'):")
repo_name = 'https://api.github.com/repos/' + name_input
projectNameList = repo_name.split("/")
projectName = projectNameList[-1]

today_str = datetime.date.today().strftime('%Y%m%d')
dataFile = projectName + '_data_' + today_str + '.txt'

# Import the data file into variable 'd'
with open(dataFile) as json_data:
    d = json.load(json_data)
    json_data.close()
    # pprint(d)      # If you want to print the record

# Initialize variables
```

```
data_list = []          # Store output data
num_open = 0            # Count number of open issues (to confirm total)

# Iterate through all records
for i in xrange(0, np.size(d)):
    data_row = []

    # Determine opened and closed times for this issue
    opened_time = d[i]['created_at']
    closed_time = d[i]['closed_at']

    # Append outputs to data_row
    data_row.append(opened_time)
    data_row.append(closed_time)

    # Check if the issue has been not closed
    if(closed_time == None):  #
        num_open += 1         # Update count of number of open issues
        data_row.append(None)
    else:
        # Create datetime objects from time stamp strings
        A = datetime.datetime.strptime(d[i]['created_at'], "%Y-%m-%dT%H:%M:%SZ")
        B = datetime.datetime.strptime(d[i]['closed_at'], "%Y-%m-%dT%H:%M:%SZ")

        # Compute duration of how long the issue remained open
        duration = B - A

        # Append duration data to row element
        data_row.append(int(duration.total_seconds()))

    # Append data_row to the data_list
    data_list.append(data_row)

# Write data elements to csv file
with open(projectName + '_issueTimes_' + today_str + '.csv', 'w') as fp:
    a = csv.writer(fp, delimiter=',')
    a.writerows(data_list)
```

List of References

[1] K. Noyes. (2010). Linux is on the rise for business. [Online]. Available: http://www.pcworld.com/article/207479/linux_on_the_rise_for_business.html

[2] L. Orsini. (2015). An open-source standard could spur drone development. [Online]. Available: http://readwrite.com/2014/10/31/linux-foundation-drone-open-source

[3] R. N. Charette. (2009). This car runs on code. [Online]. Available: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[4] J. Cheng. (2014). Army lab to provide software analysis for Joint Strike Fighter. [Online]. Available: http://defensesystems.com/Articles/2014/08/14/Army-F-35-Joint-Strike-Fighter-software-tests.aspx?p=1

[5] DOD Washington D.C., "Clarifying guidance regarding open source software," Washington, 2009.

[6] Merriam-Webster.com. Software. [Online]. Available: http://www.merriam-webster.com/dictionary/software

[7] J. Bersin. (2013). The software economy : why software jobs are taking over. [Online]. Available: http://www.forbes.com/sites/joshbersin/2013/08/05/the-software-economy-why-software-jobs-are-taking-over/print/

[8] D. Seetharaman and T. Gaynor, "Calling all ' code -aholics ': U . S . automakers vie for tech talent," 2013. [Online]. Available: http://www.dailystar.com.lb/Business/International/2013/Jul-30/225547-calling-all-code-aholics-us-automakers-vie-for-tech-talent.ashx

[9] M. Sullivan, "F-35 Joint Strike Fighter: problems completing software testing may hinder delivery of expected warfighting capabilities," no. March, 2014.

[10] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, July 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=567793.567795

[11] R. Sabhlok. (2013). Open source software: the hidden cost of free. [Online]. Available: http://www.forbes.com/sites/rajsabhlok/2013/07/18/open-source-software-the-hidden-cost-of-free/

[12] The MITRE Corporation, "Use of free and open-source software (FOSS) in the U.S. department of defense," pp. 1–168, 2003.

[13] P. Koltun, "Free and open source software use: Benefits and compliance obligations," *CrossTalk*, vol. 24, no. 6, pp. 28–30, 2011. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-81555199735&partnerID=tZOtx3y1

[14] Black Duck Software, "Future of open source software," Black Duck Software, Tech. Rep., 2015. [Online]. Available: http://www.slideshare.net/mjskok/2012-future-of-open-source-6th-annual-survey-results

[15] OSI. (2015). Open Source Initiative. [Online]. Available: http://opensource.org/about

[16] B. Duck. (2015). Black Duck Software. [Online]. Available: https://www.blackducksoftware.com/resources/data/top-20-open-source-licenses

[17] Netcraft. (2015). March 2015 web survey. [Online]. Available: http://news.netcraft.com/archives/category/web-server-survey/

[18] ScriptRock. (2015). IIS vs Apache. [Online]. Available: http://www.scriptrock.com/articles/iis-apache

[19] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm," *ICIS '00 Proceedings of the twenty first international conference on Information systems*, pp. 58–69, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=359723

[20] S. Acuna, J. W. J. Castro, O. Dieste, and N. Juristo, "A systematic mapping study on the open source software development process," *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, pp. 42–46, 2012. [Online]. Available: http://digital-library.theiet.org/content/conferences/10.1049/ic.2012.0005

[21] DOD Washington D.C. (2015). DoD open source software FAQ. [Online]. Available: http://dodcio.defense.gov/opensourcesoftwarefaq.aspx#General_information_about_OSS

[22] K. Kishida and Y. Ye, "Toward an understanding of the motivation of open source software developers," in *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR: IEEE, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1201220

[23] M. Sherman. (2011). Careers 2.0 now does GitHub. [Online]. Available: http://blog.stackoverflow.com/2011/03/careers-2-0-now-does-github/

[24] Dronecode Project. (2015). Linux Foundation collaborative projects. [Online]. Available: https://www.dronecode.org/

[25] G. Moody. (2014). Linux Foundation does open source drones. [Online]. Available: http://www.computerworlduk.com/blogs/open-enterprise/open-source-drones-3580191/

[26] E. Haruvy, F. Wu, and S. Chakravarty, "Incentives for developer's contributions and product performance metrics in open source development: an empirical exploration," Tech. Rep., 2005. [Online]. Available: http://115.111.81.83:8080/xmlui/handle/123456789/6277

[27] Stack Exchange Inc. (2015). Stack overflow. [Online]. Available: http://stackoverflow.com/

[28] J. Kahn. (2013). Open source as a civic duty. [Online]. Available: http://opensource.com/life/13/8/open-source-civic-duty

[29] L. Morgan and P. Finnegan, "Benefits and drawbacks of open source software: An exploratory study of secondary software firms," *IFIP International Federation for Information Processing*, vol. 234, pp. 307–312, 2007.

[30] Apache. (2015). OpenOffice. [Online]. Available: https://www.openoffice.org/

[31] LibreOffice. (2015). LibreOffice. [Online]. Available: https://www.libreoffice.org/

[32] D. Tuma, "Open source software : opportunities and challenges," *CrossTalk*, no. January, pp. 6–10, 2005.

[33] D. A. Wheeler. (2015). Why open source software/free software (OSS/FS, FLOSS, or FOSS)? Look at the numbers! [Online]. Available: http://www.dwheeler.com/oss_fs_why.html

[34] R. Glass, *Perspectives on Free and Open Source Software*. Cambridge, MA: The MIT Press, 2005.

[35] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-sou software products search Abstract," *Poista*, vol. 30, no. 4, pp. 246–256, 2004.

[36] S. Dean. (2014). It's 2014, and open source documentation is still lacking. [Online]. Available: http://ostatic.com/blog/its-2014-and-open-source-documentation-is-still-lacking

[37] W. Wagner, *Lightning Bugs and Other Reconnaissance Drones*. Aero Publishers, 1982.

[38] A. Roberts. (2013). By the numbers: drones. [Online]. Available: http://www.cnn.com/2013/04/05/politics/drones-btn/

[39] M. Ballve. (2015). The drones report: market forecasts, regulatory barriers, top vendors, and leading commercial applications. [Online]. Available: http://www.businessinsider.com/uav-or-commercial-drone-market-forecast-2015-2

[40] International Civil Aviation Organization, *Unmanned Aircraft Systems*. ICAO, 2009, vol. 23, no. 2. [Online]. Available: http://dx.doi.org/10.1016/B978-0-12-374518-7.00016-X

[41] D. Werner. (2013). Drone swarm: networks of small UAVs offer big capabilities. [Online]. Available: http://archive.defensenews.com/article/20130612/C4ISR/306120029/

[42] Federal Aviation Administration. (2015). Unmanned Aircraft Systems. [Online]. Available: https://www.faa.gov/uas/

[43] T. Chung, "Countering adversarial unmanned systems," unpublished.

[44] M. Bronz, J. M. Moschetta, P. Brisset, and M. Gorez, "Towards a long endurance MAV," *International Jounral of Micro Air Vehicles*, vol. 1, no. 4, pp. 244–245, 2009.

[45] OSI. (2015). Licenses and Standards. [Online]. Available: http://opensource.org/licenses

[46] R. E. Ball, "Southeast Asia Conflict, 1964-1973," in *The Fundamentals of Aircraft Combat Survivability Analysis and Design, Second Edition*, 2nd ed., J. A. Schetz, Ed. American Institute of Aeronautics and Astronautics, Inc., 2003, ch. 1, pp. 95–101.

[47] S. Rousseau. (2003). Laser-guided missiles. [Online]. Available: http://www3.nd.edu/~techrev/Archive/Spring2002/a9.html

[48] Defense Acquisition University. (2014). Live fire test and evaluation. [Online]. Available: https://dap.dau.mil/acquipedia/Pages/ArticleDetails.aspx?aid=91336359-3460-445d-9ede-cb4876a4e135

[49] J. L. Lions, "Ariane 5 Flight 501 Failure," Paris, Tech. Rep., 1996. [Online]. Available: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

[50] N. Barry. (2013). All bugs have software. [Online]. Available: http://nathanbarry.com/bugs-software/

[51] E. Limer. (2014). Heartbleed: why the internet's gaping security hole is so scary. [Online]. Available: http://gizmodo.com/heartbleed-why-the-internets-gaping-security-hole-is-1560812671

[52] R. E. Ball, *The Fundamentals of Aircraft Survivability, Analysis and Design*, J. A. Schetz, Ed. AIAA, 2003.

[53] Gitorious. (2015). FlightGear. [Online]. Available: https://www.gitorious.org/fg

[54] Google. (2015). FlightGear - bugs. [Online]. Available: https://code.google.com/hosting/moved?project=flightgear-bugs

[55] OpenHub. (2015). FlightGear Statistics. [Online]. Available: https://www.openhub.net/p/flightgear

[56] Defense Information Systems Agency. (2015). Forge.mil. [Online]. Available: http://forge.mil/

[57] Github Inc. (2015). Github developer: API. [Online]. Available: https://developer.github.com/v3/

[58] K. Reitz. (2015). Requests: HTTP for humans. [Online]. Available: http://docs.python-requests.org/en/latest/

[59] Python Software Foundation. (2015). CSV file reading and writing. [Online]. Available: https://docs.python.org/2/library/csv.html

[60] OpenHub. (2015). ardupilot. [Online]. Available: https://www.openhub.net/p/ardupilot

[61] OpenHub. (2015). MAVLink. [Online]. Available: https://www.openhub.net/p/mavlink

[62] OpenHub. (2015). MAVLink project summary. [Online]. Available: https://www.openhub.net/p/mavlink

[63] C. Kolassa, D. Riehle, and M. a. Salim, "The empirical commit frequency distribution of open source projects," *Proceedings of the 9th International Symposium on Open Collaboration - WikiSym '13*, pp. 1–8, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491055.2491073

[64] J. Bowes. (2012). A code hosting comparison for open source projects. [Online]. Available: http://opensource.com/life/12/11/code-hosting-comparison

[65] Canonical LTD. (2015). Launchpad. [Online]. Available: https://launchpad.net/

[66] CollabNet Inc. (2015). CloudForge.org. [Online]. Available: https://cloudforge.com/

[67] G. Stansberry. (2008). 7 version control systems reviewed. [Online]. Available: http://www.smashingmagazine.com/2008/09/18/the-top-7-open-source-version-control-systems/

[68] MITRE. (2015). Common vulnerabilities and exposures. [Online]. Available: https://cve.mitre.org/

[69] C. Adams, "Threats and threat effects," unpublished.

[70] R. E. Ball, *The Fundamentals of Aircraft Combat Survivability, Analysis and Design*, 2nd ed., J. A. Schetz, Ed. AIAA, 2003.

[71] J. J. H. Piggott, "Open source software attributes as success indicators," 2013.

[72] D. A. Wheeler. (2011). How to evaluate open source software / free software programs. [Online]. Available: http://www.dwheeler.com/oss_fs_eval.html

[73] E. Petrinja, R. Nambakam, and A. Sillitti, "Introducing the opensource maturity model," *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, FLOSS 2009*, pp. 37–41, 2009.

[74] Qualipso. (2008). Trust and quality on free and open source systems. [Online]. Available: http://qualipso.icmc.usp.br/OMM/

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California